UNIT V

PERL

Perl backgrounder – Perl overview – Perl parsing rules – Variables and Data – Statements and Control structures – Subroutines, Packages, and Modules- Working with Files –Data Manipulation.

Introduction to Perl:-

What is Perl?

Perl is a programming language. Perl stands for Practical Report and Extraction Language. You'll notice people refer to 'perl' and "Perl". "Perl" is the programming language as a whole whereas 'perl' is the name of the core executable. There is no language called "Perl5" -- that just means "Perl version 5". Versions of Perl prior to 5 are very old and very unsupported.

Some of Perl's many strengths are:

Speed of development. You edit a text file, and just run it. You can develop programs very quickly like this. No separate compiler needed. I find Perl runs a program quicker than Java, let alone compare the complete modify-compile-run-oh-no-forgot-that-semicolon sequence.

Power. Perl's regular expressions are some of the best available. You can work with objects, sockets...everything a systems administrator could want. And that's just the standard distribution. Add the wealth of modules available on CPAN and you have it all. Don't equate scripting languages with toy languages.

Usuability. All that power and capability can be learnt in easy stages. If you can write a batch file you can program Perl. You don't have to learn object oriented programming, but you can write OO programs in Perl. If autoincrementing non-existent variables scares you, make perl refuse to let you. There is always more than one way to do it in Perl. You decide your style of programming, and Perl will accommodate you.

Portability. On the Superhighway to the Portability Panacea, Perl's Porsche powers past Java's jaded jalopy. Many people develop Perl scripts on NT, or Win95, then just FTP them to a Unix server where they run. No modification necessary.

Editing tools You don't need the latest Integrated Development Environment for Perl. You can develop Perl scripts with any text editor. Notepad, vi, MS Word 97, or even direct off the console. Of course, you can make things easy and use one of the many freeware or shareware programmer's file editors.

Price. Yes, 0 guilders, pounds, dmarks, dollars or whatever. And the peer to peer support is also free, and often far better than you'd ever get by paying some company to answer the phone and tell you to do what you just tried several times already, then look up the same reference books you already own.

What can I do with Perl ?

Just two popular examples :

The Internet

Go surf. Notice how many websites have dynamic pages with .pl or similar as the filename extension? That's Perl. It is the most popular language for CGI programming for many reasons, most of which are mentioned above. In fact, there are a great many more dynamic pages written with perl that may not have a .pl extension. If you code in Active Server Pages, then you should try using ActiveState's PerlScript. Quite frankly, coding in PerlScript rather than VBScript or JScript is like driving a car as opposed to riding a bicycle. Perl powers a good deal of the Internet.

Systems Administration

If you are a Unix sysadmin you'll know about sed, awk and shell scripts. Perl can do everything they can do and far more besides. Furthermore, Perl does it much more efficiently and portably. Don't take my word for it, ask around.

If you are an NT sysadmin, chances are you aren't used to programming. In which case, the advantages of Perl may not be clear. Do you need it? Is it worth it?

A few examples of how I use Perl to ease NT sysadmin life:

User account creation. If you have a text file with the user's names in it, that is all you need. Create usernames automatically, generate a unique password for each one and create the account, plus create and share the home directory, and set the permissions.

Event log munging. NT has great Event Logging. Not so great Event Reading. You can use Perl to create reports on the event logs from multiple NT servers.

Anything else that you would have used a batch file for, or wished that you could automate somehow. Now you *can*.

Perl Backgrounder :

Versions and Naming Conventions

Variables:

- single word: atom, chain
- multi word: atomName, centralAtomName
- constants: CALPHA_ATOM_NAME or ATOM_NAME_CALPHA, ATOM_NAME_CBETA

Perl is not type-safe and this can cause confusion and errors. Use a limited prefix notation for such common basic types as array, hash, FileHandle.

- array refs ('a' prefix): aAtoms, aChains
- hash refs ('h' prefix): hNames2Places, hChains
- FileHandle objects ('fh' prefix): fhIn, fhOut, fhPdb
- or ("ist"=input stream, "ost"=output stream): ostPdb, istMsa

Functions

- single word: Trim()
- multi word: OpenFilesForReading()

Modules (packages that are not classes)

- single word: Assert
- multi word: FileIoHelper

Classes

As for modules but with 'C' prefix: CStopwatch, CWindowPanel, Pdb::CResidue

Instance methods

- public method: plot(), getColour(), classifyHetGroups()
- private method: _plot(), _getColour(), _classifyHetGroups()
- accessor methods same as JavaBeans: getProperty(), setProperty(), isProperty()

Perl, perl or PeRl?

There is also a certain amount of confusion regarding the capitalization of Perl. Should it be written Perl or perl? Larry Wall now uses "Perl" to signify the language proper and "perl" to signify the implementation of the language.

Perl History

Version	Date	Version Details
Perl 0		Introduced Perl to Larry Wall's office associates
Perl 1	Jan 1988	Introduced Perl to the world
Perl 2	Jun 1988	Introduced Harry Spencer's regular expression package
Perl 4	Mar 1991	Introduced the first "Camel" book (Programming O'Reilly & Associates). The book drove the name change, just so it could refer to Perl 4, instead of Perl 3.
Perl 4.036	Feb 1993	The last stable release of Perl 4
Perl 5	Oct 1994	The first stable release of Perl 5, which introduced a number of new features and a complete rewrite.
Perl 5.005_02	Aug 1998	The next major stable release
Perl 5.005_03	Mar 1999	The last stable release before 5.6
Perl 5.6	Mar 2000	Introduced unified **fork** support, better threading, an updated Perl compiler, and the **our** keyword

Main Perl Features:

a)Perl Is Free : Perl's source code is open and free anybody can download the C source that constitutes a Perl interpreter. Furthermore, you can easily extend the core functionality of Perl both within the realms of the interpreted language and by modifying the Perl source code.

b) Perl Is Simple to Learn, Concise, and Easy to Read:

It has a syntax similar to C and shell script, among others, but with a less restrictive format. Most programs are quicker to write in Perl because of its use of built-in functions and a huge standard and contributed library. Most programs are also quicker to execute than other languages because of Perl's internal architecture.

c) Perl Is Fast :

Compared to most scripting languages, this makes execution almost as fast as compiled C code. But, because the code is still interpreted, there is no compilation process, and applications can be written and edited much faster than with other languages, without any of the performance problems normally associated with an interpreted language.

d) Perl Is Extensible:

You can write Perl-based packages and modules that extend the functionality of the language. You can also call external C code directly from Perl to extend the functionality. **e) Perl Has Flexible Data Types:**

You can create simple variables that contain text or numbers, and Perl will treat the variable data accordingly at the time it is used.

f) Perl Is Object Oriented:

Perl supports all of the object-oriented features—inheritance, polymorphism, and encapsulation.

g)Perl Is Collaborative:

There is a huge network of Perl programmers worldwide. Most programmers supply, and use, the modules and scripts available via CPAN, the Comprehensive Perl Archive Network.

Perl Overview:

Installing and using Perl

Perl was developed by Larry Wall. It started out as a scripting language to supplement rn, the USENET reader. It available on virtually every computer platform.

Perl is an interpreted language that is optimized for string manipulation, I/O, and system tasks. It has built in for most of the functions in section 2 of the UNIX manuals -- very popular with sys administrators. it incorporates syntax elements from the Bourne shell, csh, awk, sed, grep, and C. It provides a quick and effective way to write interactive web applications

Writing a Perl Script

Perl scripts are just text files, so in order to actually "write" the script, all you need to do is create a text file using your favorite text editor. Once you've written the script, you tell Perl to execute the text file you created.

Under Unix, you would use

```
$ perl myscript.pl
```

and the same works under Windows:

```
C:\> perl myscript.pl
```

Under Mac OS, you need to drag and drop the file onto the MacPerl application.

Perl scripts have a .pl extension, even under Mac OS and Unix.

Perl Under Unix

The easiest way to install Perl modules on Unix is to use the CPAN module. For example:

```
shell> perl -MCPAN -e shell
cpan> install DBI
cpan> install DBD::mysql
```

The DBD::mysql installation runs a number of tests. These tests attempt to connect to the local MySQL server using the default user name and password. (The default user name is your login name on Unix, and ODBC on Windows. The default password is "no password.") If you cannot connect to the server with those values (for example, if your account has a password), the tests fail. You can use force install DBD::mysql to ignore the failed tests.

DBI requires the Data::Dumper module. It may be installed; if not, you should install it before installing DBI.

Perl Under Windows

1. Log on to the Web server computer as an administrator.
2. Download the ActivePerl installer from the following ActiveState Web site:
   http://www.activestate.com/ (http://www.activestate.com/)
3. Double-click the **ActivePerl** installer.
4. After the installer confirms the version of ActivePerl that it is going to be installed, click **Next.**
5. If you agree with the terms of the license agreement, click **I accept the terms in the license agreement,**and then Click **Next.**

Click **Cancel** if you do not accept the license agreement. If you do so, you cannot continue the installation.

6. To install the whole ActivePerl distribution package (this step is recommended), click **Next** to continue the installation.

The software is installed in the default location (typically C:\Perl).

7. To customize the individual components or to change the installation folder, follow the instructions that appear on the screen.
8. When you are prompted to confirm the addition features that you want to configure during the installation, click any of the following settings, and then click **Next:**
   - **Add Perl to the PATH environment variable**: Click this setting if you want to use Perl in a command prompt without requiring the full path to the Perl interpreter.
   - **Create Perl file extension association:** Click this setting if you want to allow Perl scripts to be automatically run when you use a file that has the Perl file name extension (.pl) as a command name.
   - **Create IIS script mapping for Perl:** Click this setting to configure IIS to identify Perl scripts as executable CGI programs according to their file name extension.
   - **Create IIS script mapping for Perl ISAPI:** Click this setting to use Perl scripts as an ISAPI filter.
9. Click **Install** to start the installation process.
10. After the installation has completed, click **Finish.**

Perl Syntax and Parsing Rules:

The Perl parser thinks about all of the following when it looks at a source line:

Basic syntax The core layout, line termination, and so on

Comments If a comment is included, ignore it

Component identity Individual terms (variables, functions and numerical and textual constants) are identified

Bare words Character strings that are not identified as valid terms

Precedence Once the individual items are identified, the parser processes the statements according to the precedence rules, which apply to all operators and terms

Context What is the context of the statement, are we expecting a list or scalar, a number or a string, and so on. This actually happens during the evaluation of individual elements of a line, which is why we can nest functions such as sort, reverse, and keys into a single statement line

Logic Syntax For logic operations, the parser must treat different values, whether constant- or variable-based, as true or false values.

All of these present some fairly basic and fundamental rules about how Perl looks at an entire script.

Basic Syntax

The basic rules govern such things as line termination and the treatment of white space. These basic rules are

Lines must start with a token that does not expect a left operand

Lines must be terminated with a semicolon, except when it's the last line of a block, where the semicolon can be omitted.

White space is only required between tokens that would otherwise be confusing, so spaces, tabs, newlines, and comments (which Perl treats as white space) are ignored.

The line

```
sub menu{print"menu"}
```

works as it would if it were more neatly spaced.

Lines may be split at any point, providing the split is logically between two tokens.

Component Identity : When Perl fails to identify an item as one of the predefined operators, it treats the character sequence as a "term." Terms are core parts of the Perl language and include variables, functions, and quotes. The term-recognition system uses these rules:

Variables can start with a letter, number, or underscore, providing they follow a suitable variable character, such as $, @, or %.

Variables that start with a letter or underscore can contain any further combination of letters, numbers, and underscore characters.

Variables that start with a number can only consist of further numbers—be wary of using variable names starting with digits. The variables such as $0 through to $9 are used for group matches in regular expressions.

Subroutines can only start with an underscore or letter, but can then

contain any combination of letters, numbers, and underscore characters. Case is significant—$VAR, $Var, and $var are all different variables.

Each of the three main variable types have their own name space—$var, @var, and %var are all separate variables.

Filehandles should use all uppercase characters—this is only a convention, not a rule, but it is useful for identification purposes.

Variables and Data:

Perl has three built in variable types:

Scalar

Array

Hash

Basic Naming Rules The basic rules that apply to the naming of variables within Perl:

Variable names can start with a letter, a number, or an underscore, although they normally begin with a letter and can then be composed of any combination of letters, numbers, and the underscore character.

Variables can start with a number, but they must be entirely composed of that number; for example, $123 is valid, but $1var is not.

Variable names that start with anything other than a letter, digit, or underscore are generally reserved for special use by Perl (see "Special Variables" later in this chapter).

Variable names are case sensitive; $foo, $FOO, and $fOo are all separate variables as far as Perl is concerned.

As an unwritten (and therefore unenforced) rule, names all in uppercase are constants.

All scalar values start with $, including those accessed from an array of hash, for example $array[0] or $hash{key}.

All array values start with @, including arrays or hashes accessed in slices, for example @array[3..5,7,9] or @hash{'bob', 'alice'}. All hashes start with %.

Namespaces are separate for each variable type—the variables $var, @var, and %var are all different variables in their own right.

In situations where a variable name might get confused with other data (such as when embedded within a string), you can use braces to quote the name. For example, ${name}, or %{hash}.

Scalar Variables:

A scalar represents a single value as follows:

```
my $animal = "camel"; my $answer = 42;
```

A scalar values can be strings, integers or floating point numbers, and Perl will automatically convert between them as required. There is no need to pre-declare your variable types. Scalar values can be used in various ways:

```
print $animal;
print "The animal is $animal\n";
```

```
print "The square of $answer is ", $answer * $answer, "\n";
```

There are a number of "magic" scalars with names that look like punctuation or line noise. These special variables are used for all kinds of purposes and they wioll be discussed in Special Variables sections. The only one you need to know about for now is $_ which is the "default variable".

```
print;      # prints contents of $_ by default
```

Literals :

Literal is a value that is represented "as is" or hard-coded in your source code. When you see the four characters 45.5 in programs it really refers to a value of forty-five and a half. Perl uses four types of literals. Here is a quick glimpse at them:

Numbers - This is the most basic data type.

Strings - A string is a series of characters that are handled as one unit.

Arrays - An array is a series of numbers and strings handled as a unit. You can also think of an array as a list.

Associative Arrays - This is the most complicated data type. Think of it as a list in which every value has an associated lookup item

Numeric Literals:

Numeric literals are simply constant numbers. Numeric literals are much easier to comprehend and use than string literals. Thcrc arc only a few basic ways to express numeric literals.

a) String Literals

i)Single-Quoted-Strings:

Single quoted are a sequence of characters that begin and end with a single quote. These quotes are not a part of the string they just mark the beginning and end for the Perl interpreter. If you want a ' inside of your string you need to preclude it with a \ like this \' as you'll see below.

```
'four'      #has four letters in the string
'can\'t'    #has five characters and represents "can't"
'hi\there'  #has eight characters and represents"hi\\there" (one \ in the string)
'blah\\blah'.   #has nine characters and represents "blah\\blah" (one \ in the string)
```

ii)Double-Quoted-Strings:

Double quoted strings act more like strings in C or C++ the backslash allows you to represent control characters. Another nice feature Double-Quoted strings offers is variable interpolation this substitutes the value of a variable into the string.

Some of the things you can put in a Double-Quoted String

Representation	What it Means
\a	Bell
\b	Backspace
\e	Escape
\f	Formfeed
\n	Newline

\r	Return
\t	Tab
\\	Backslash
\"	Double quote
\007	octal ascii value this time 007
\x07	hex ascii value this time 007 or
\cD	any control character.. here it is control-D
\l	lowercase next letter
\u	uppercase next letter
\L	lowercase all letters until \E
\U	uppercase all letters until \E
\Q	Backslash quote all nonletters and nonnumbers until \E
\E	Stop \U \L or \Q

Statements and Control Structures:

Code Blocks:

A sequence of statements is called a code block, or simply just a block. A block of code could be any of the code examples that we have seen thus far. The only difference is, to make them a block, we would surround them with {}.

```
use strict;
{
        my    $var;
        Statement;
        Statement;
        Statement;
}
```


Anything that looks like that is a block. Blocks are very simple, and are much like code blocks in languages like C, C++, and Java. However, in Perl, code blocks are decoupled from any particular control structure. The above code example is a valid piece of Perl code that can appear just about anywhere in a Perl program. Of course, it is only particularly useful for those functions and structures that use blocks.

Conditional Statements:

The conditional statements are **if** and **unless**, and they allow you to control the execution of your script. There are five different formats for the **if** statement:

```
if (EXPR)
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
STATEMENT if (EXPR)
```

The first format is classed as a simple statement, since it can be used at the end of another statement without requiring a block, as in:print "Happy Birthday!\n" if ($date == $today);

In this instance, the message will only be printed if the expression evaluates to a true value.

The second format is the more familiar conditional statement that you may have come across in other languages:

```
if ($date == $today)
{
  print "Happy Birthday!\n";
}
```

This produces the same result as the previous example.

The third format allows for exceptions. If the expression evaluates to true, then the first block is executed; otherwise (else), the second block is executed:

```
if ($date == $today)
{
  print "Happy Birthday!\n";
}
else
{
  print "Happy Unbirthday!\n";
}
```

The fourth form allows for additional tests if the first expression does not return true. The elsif can be repeated an infinite number of times to test as many different alternatives as are required:

```
if ($date == $today)
{
  print "Happy Birthday!\n";
}
elsif ($date == $christmas)
{
  print "Happy Christmas!\n";
}
```

The fifth form allows for both additional tests and a final exception if all the other tests fail:

```
if ($date == $today)
{
  print "Happy Birthday!\n";
}
elsif ($date == $christmas)
{
```

```
  print "Happy Christmas!\n";
}else
{
  print "Happy Unbirthday!\n";
}
```

The unless statement automatically implies the logical opposite of **if**, so unless the **EXPR** is true, execute the block. This means that the statement print "Happy Unbirthday!\n" unless ($date == $today);is equivalent to print "Happy Unbirthday!\n" if ($date != $today);

For example, the following is a less elegant solution to the preceding if...else. Although it achieves the same result, example:

```
unless ($date != $today)
{
  print "Happy Unbirthday!\n";
}
else
{
  print "Happy Birthday!\n";
}
```

The final conditional statement is actually an operator.the conditional operator. It is synonymous with the if...else conditional statement but is shorter and more compact. The format for the operator is: (expression) ? (statement if true) : (statement if false)

For example, we can emulate the previous example as follows: $date == $today) ? print "Happy B.Day!\n" : print "Happy Day!\n";

Loops:

Perl supports four main loop types:

1. while
2. for
3. until
4. foreach

In each case, the execution of the loop continues until the evaluation of the supplied expression changes.

In the case of a **while** loop execution continues while the expression evaluates to true.

The until loop executes while the loop expression is false and only stops when the expression evaluates to a true value.

The list forms of the **for** and **foreach** loop are special cases. They continue until the end of the supplied list is reached.

while Loops

The while loop has three forms:

while EXPRLABEL

while (EXPR) BLOCKLABEL

while (EXPR) BLOCK continue BLOCK

In first form, the expression is evaluated first, and then the statement to which it applies is evaluated. For example, the following line increases the value of $linecount as long as we continue to read lines from a given file:

```
$linecount++ while ();
```

To create a loop that executes statements first, and then tests an expression, you need to combine while with a preceding **do** {} statement. For example:

```
Do
{
  $calc += ($fact*$ivalue);
} while $calc <100;
```

In this case, the code block is executed first, and the conditional expression is only evaluated at the end of each loop iteration.

The second two forms of the while loop repeatedly execute the code block as long as the result from the conditional expression is true. For example, you could rewrite the preceding example as:

```
while($calc < 100)
{
  $calc += ($fact*$ivalue);
}
```

until Loops

The inverse of the while loop is the until loop, which evaluates the conditional expression and reiterates over the loop only when the expression returns false. Once the expression returns true, the loop ends.

In the case of a do.until loop, the conditional expression is only evaluated at the end of the code block. In an until (EXPR) BLOCK loop, the expression is evaluated before the block executes. Using an until loop, you could rewrite the previous example as:

```
Do
{
 $calc += ($fact*$ivalue);
} until $calc >= 100;
```

This is equivalent to

```
do
{
 $calc += ($fact*$ivalue);
} while $calc <100;
```

for Loops

A for loop is basically a while loop with an additional expression used to reevaluate the original conditional expression. The basic format is:

LABEL for (EXPR; EXPR; EXPR) BLOCK

The first EXPR is the initialization - the value of the variables before the loop starts iterating. The second is the expression to be executed for each iteration of the loop as a test. The third expression is executed for each iteration and should be a modifier for the loop variables.

Thus, you can write a loop to iterate 100 times like this:

```
for ($i=0;$i<100;$i++)
{
...
}
```

You can place multiple variables into the expressions using the standard list operator (the comma):

```
for ($i=0, $j=0;$i<100;$i++,$j++)
```

You can create an infinite loop like this:

```
for(;;)
{
...
}
```

foreach Loops

he last loop type is the foreach loop, which has a format like this:

LABEL foreach VAR (LIST) BLOCK

LABEL foreach VAR (LIST) BLOCK continue BLOCK

Using a for loop, you can iterate through the list using:

```
for ($index=0;$index<=@months;$index++)
{
  print "$months[$index]\n";
}
```


This is messy, because you.re manually selecting the individual elements from the array and using an additional variable, $index, to extract the information. Using a foreach loop, you can simplify the process:

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

The foreach loop can even be used to iterate through a hash, providing you return the list of values or keys from the hash as the list:

```
foreach $key (keys %monthstonum)
{
```

```
  print "Month $monthstonum{$key} is $key\n";
}
```

The continue Block:

The **continue** block is executed immediately after the main block and is primarily used as a method for executing a given statement (or statements) for each iteration, irrespective of how the current iteration terminated. It is somehow equivalent to for loop

```
{
  my $i = 0;
  while ($i <100)
  { ... }
  continue
  {
    $i++;
  }
}
```

This is equivalent to

```
for (my $i = 0; $i < 100; $i++)
{ ... }
```

Labels:

Labels can be applied to any block, but they make the most sense on loops. By giving your loop a name, you allow the loop control keywords to specify which loop their operation should be applied to. The format for a labeled loop is:

```
LABEL: loop (EXPR) BLOCK ...
```

For example, to label a for loop:

```
ITERATE: for (my $i=1; $i<100; $i++)
{
  print "Count: $i\n";
}
```

Loop Control:

There are three loop control keywords: next, last, and redo.

The **next** keyword skips the remainder of the code block, forcing the loop to proceed to the next value in the loop. For example:

```
while (<DATA>)
{
  next if /^#/;
}
```

Above code would skip lines from the file if they started with a hash symbol. If there is a **continue** block, it is executed before execution proceeds to the next iteration of the loop.

The **last** keyword ends the loop entirely, skipping the remaining statements in the code block, as well as dropping out of the loop. The last keyword is therefore identical to the break keyword in C and Shellscript. For example:

```
while ()
{
  last if ($found);
}
```

Would exit the loop if the value of $found was true, whether the end of the file had actually been reached or not. The **continue** block is not executed.

The **redo** keyword reexecutes the code block without reevaluating the conditional statement for the loop. This skips the remainder of the code block and also the **continue** block before the main code block is reexecuted. For example, the following code would read the next line from a file if the current line terminates with a backslash:

```
while(<DATA>)
{
  if (s#\\$#)
  {
    $_ .= <DATA>;
    redo;
  }
}
```

Here is an example showing how labels are used in inner and outer loops

```
OUTER:
  while(<DATA>)
  {
    chomp;
    @linearray=split;
    foreach $word (@linearray)
    {
      next OUTER if ($word =~ /next/i)
    }
  }
```


goto

There are three basic forms: **goto LABEL, goto EXPR,** and **goto &NAME.** In each case, execution is moved from the current location to the destination.

In the case of goto **LABEL,** execution stops at the current point and resumes at the point of the label specified.

The goto &NAME statement is more complex. It allows you to replace the currently executing subroutine with a call to the specified subroutine instead.

Subroutines, Packages, and Modules

Subroutines:

a)Functions:

Definition:

Function is a block of source code which does one or some tasks with specified purpose.

Advantages:

It reduces the Complexity in a program by reducing the code.

It also reduces the Time to run a program.In other way,It's directly proportional to Complexity.

It's easy to find-out the errors due to the blocks made as function definition outside the main function.

Subroutines, like variables, can be declared and defined.

Various forms:

```
sub NAME
sub NAME PROTO
sub NAME ATTRS
sub NAME PROTO ATTRS
```

where as

PROTO – Prototype, ATTRS – Attributes

Examples:

```
sub message
{
        print "Hello!\n";
}
```


To call the subroutine and get a result;

```
add(1,2);
```

b) Arguments:

In perl we do not declare arguments a function takes while we defining a function, we still may pass arguments to a Perl function. All arguments are passed into a function through the special array @_. Thus, we can send as many arguments as we want and also function may receive different number of arguments.

Example 1:

simple function adds two numbers and prints the result:

```
sub add
{
        $result = $_[0] + $_[1];
        print "The result was: $result\n";
}
```

To call the subroutine and get a result,

```
add(1,2);
```

Example 2:

```
submy_args
{
my $i;
print "My arguments:\n";
for($i=0;$i<=$#_;$i++){
print "Argument $i: $_[$i]\n";
  }
print "\n";
 }
 my $name = "arg one";
 my @names = ('This', 'is', 'an', 'array');
 my_args($name);
 my_args($name, 123);
 my_args(123, 345, @names, $name);
```

Counting Arguments:

If you want to count the number of arguments that you have received, just access the @_ in a scalar context:

Example:

```
my $count = @_;
```

c) Return Values:

A subroutine is always part of some expression. The value of the subroutine invocation is called the return value. The return value of a subroutine is the value of the return statement or of the last expression evaluated in the subroutine.

For example, let's define this subroutine:

```
subsum_of_a_and_b
{
    return $a + $b;
}
```

The last expression evaluated in the body of this is the sum of $a and $b , so the sum of $a and $b will be the return value. Here's that in action:

```
$a = 3;
$b = 4;
$c = sum_of_a_and_b();
# $c gets 7
$d = 3 * sum_of_a_and_b();
```

```
# $d gets 21
```

A subroutine can also return a list of values when evaluated in a list context. Consider this subroutine and invocation:

```
sublist_of_a_and_b
{
return($a,$b);
}
$a = 5;
$b = 6;
@c = list_of_a_and_b();
# @c gets (5,6)
```

The last expression evaluated really means the last expression evaluated, rather than the last expression defined in the body of the subroutine. For example, this subroutine returns $a if $a >0 ; otherwise it returns $b :

d) Error Notification:

The easiest way to report an error within a subroutine is to return the undefined value undef function.

i) undef:

Undefines the value of EXPR. Use on a scalar, list, hash, function, or typeglob. Use on a hash with a statement such as undef $hash{$key}; actually sets the value of the specified key to an undefined value. If you want to delete the element from the hash, use the delete function.

Syntax

undef EXPR

undef

Example

```
#!/usr/bin/perl -w
$scalar = 10;
@arrary = (1,2);
print "1 - Value of Scalar is $scalar\n";
print "1 - Value of Array is @array\n";
undef( $scalar );
undef( @array );
print "2 - Value of Scalar is $scalar\n";
print "2 - Value of Array is @array\n";
```

It will produce following results:

```
1 - Value of Scalar is 10
1 - Value of Array is
```

Use of uninitialized value in concatenation (.) or
string at test.pl line 13.
2 - Value of Scalar is
2 - Value of Array is

e) Context :

If you write a Perl program, you refer to lists with an "@" symbol in front of the list variable name. But depending on the context of how you write it, the program may interpret it as

The list contents (if the operation is normal for a list).

The length of the list (i.e. the element count) if that's the only thing that makes sense.

A space separated string with all the items in the list joined together (if it's in double quotes).

For example:

```
@salad = ("apple","banana","cherry");
$salad[3] = "tomato";
$salad[8] = "fig";
print @salad."\n"; # scalar context
print "@salad.\n"; # double quote context
```

f) will display:

```
applebananacherrytomatofig
9
apple banana cherry tomato fig.
```

g) Attributes:

Attributes allow you to add extra semantics to any Perl subroutine or variable.Currently Perl supports only three attributes: locked, method, and lvalue

i) The locked Attribute:

```
# Only one thread is allowed into this function.
subafunc : locked { ... }
# Only one thread is allowed into this function on a given object.
subafunc : locked method { ... }
```

Setting the locked attribute is meaningful only when the subroutine or method is intended to be called by multiple threads simultaneously. When set on a nonmethod subroutine, Perl ensures that a lock is acquired on the subroutine itself before that subroutine is entered. When set on a method subroutine (that is, one also marked with the method attribute), Perl ensures that any invocation of it implicitly locks its first argument (the object) before execution.

Semantics of this lock are the same as using the lock operator on the subroutine as the first statement in that routine.

ii) The Method attribute:

```
subafunc : method { ... }
```

Currently this has only the effect of marking the subroutine so as not to trigger the "Ambiguous call resolved as CORE::%s" warning. (We may make it mean more someday.)

The attribute system is user-extensible, letting you create your own attribute names. These new attributes must be valid as simple identifier names (without any punctuation other than the "_" character). They may have a parameter list appended, which is currently only checked for whether its parentheses nest properly.

Here are examples of valid syntax (even though the attributes are unknown):

```
subfnord (&\%) : switch(10,foo(7,3)) :  expensive;
subplugh () : Ugly('\(") :Bad;
subxyzzy : _5x5 { ... }
```

Here are examples of invalid syntax:

```
subfnord : switch(10,foo();  # ()-string not balanced
subsnoid : Ugly('(');          # ()-string not balanced
subxyzzy : 5x5;            # "5x5" not a valid identifier
subplugh : Y2::north;       # "Y2::north" not a simple identifier
subsnurt : foo + bar;       # "+" not a colon or space
```

The attribute list is passed as a list of constant strings to the code that associates them with the subroutine. Exactly how this works (or doesn't) is highly experimental. Check attributes(3) for current details on attribute lists and their manipulation.

iii) The lvalue Attribute

Using lvalue, a subroutine can be used as a modifiablescalar value. For example, you can do this:

```
mysub() = 5;
```

This is particularly useful in situations where you want to use a method on an object toaccept a setting, instead of setting the value on the object directly.

For example:

```
submysub : lvalue
{
$val;
}
```


h) Prototypes:

Perl's prototypes are not the prototypes in other languages. Other languages use prototypes to define, name and type the arguments to a subroutine. Perl prototypes are about altering the context in which the subroutine's arguments are evaluated, instead of the normal list context. Perl prototypes should **not** be used merely as a check that the right arguments have been passed in. For that use a module such as **I)Params::Validate.**

The prototype of a function can be queried by calling prototype(). Prefix built-in functions with "CORE::". prototype("CORE::abs").

The prototype function Returns a string containing the prototype of the function or reference specified by EXPR, or undef if the function has no prototype.

A subroutine's prototype is defined in its declaration, and enforces constraints on the interpretation of arguments passed to it. For example:

```
$func_prototype = prototype ( "myprint" );
print "myprint prototype is $func_prototype\n";
submyprint($$){
print "This is test\n";
}
```

It will produce following results:

Myprint prototype is $$

j) Traps:

Prototypes do not work on methods.

Prototypes do not work on function references.

Calling a function with a leading & (&foo) disables prototypes. But you shouldn't be calling functions like that.

Packages:

A package is a collection of code which lives in its own namespace. A namespace is a named collection of unique variable names (also called a symbol table).

Namespaces prevent variable name collisions between packages. Packages enable the construction of modules which, when used, won't clobber variables and functions outside of the module's own namespace.

a) The Package Statement:

package statement switches the current naming context to a specified namespace (symbol table). If the named package does not exists, a new namespace is first created.

```
$i = 1; print "$i\n"; # Prints "1" package foo;
$i = 2; print "$i\n"; # Prints "2" package main;
print "$i\n"; # Prints "1"
```


The package stays in effect until either another package statement is invoked, or until the end of the end of the current block or file.

You can explicitly refer to variables within a package using the :: package qualifier

b) Package Symbol Tables :

The symbol table is the list of active symbols (functions variables objects) within a package. Each package has its own symbol table and with some exceptions all the identifiers starting with letters or underscores are stored within the corresponding symbol table for each package. This means that all other identifiers including all of the special punctuation-only variables such as $_ are stored within the main package The symbol table for a package can be accessed as a hash. For example, the main package's symbol table can be accessed as %main:: or, more simply, as %::. Likewise, symbol tables for other packages are %MyMathLib::. The format is hierarchical, so

that symbol tables can be traversed using standard Perl code. For example

```
foreach $symname (sort keys %main::)
```

```
{
    local *symbol = $main::{$symname};
    print "$$symname is defined\n" if defined $symbol;
    print "\@$symname is defined\n" if defined @symbol;
    print "\%$symname is defined\n" if defined %symbol;
}
```

You can also use the symbol table to define static scalars by assigning a value to atype glob:

```
*C = 299792458;
```

c) Special Blocks

BEGIN and END Blocks

You may define any number of code blocks named BEGIN and END which act as constructors and destructors respectively.

```
BEGIN { ... }
END { ... }
BEGIN { ... }
END { ... }
```

Every BEGIN block is executed after the perl script is loaded and compiled but before any other statement is executed.Every END block is executed just before the perl interpreter exits.

The BEGIN and END blocks are particularly useful when creating Perl modules.

Modules:

Modules are the loadable libraries of the Perl world. A Perl module is a reusable package defined in a library file whose name is the same as the name of the package (with a .pm on the end). A Perl module is a self-contained piece of [Perl] code that can be used by a Perl program.

A Perl module file called "Foo.pm" might contain statements like this.

```
#!/usr/bin/perl
package  Foo;
sub bar
{
  print "Hello $_[0]\n"
}
sub blat {
  print "World $_[0]\n"
}
```

a) Few noteable points about modules

The functions require and use will load a module.

Both use the list of search paths in @INC to find the module (you may modify it!) Both call the eval function to process the code The 1; at the bottom causes eval to evaluate to TRUE (and thus not fail)

b) Creating Modules:

There are two ways for a module to make its interface available to your program: by exporting symbols or by allowing method calls. We'll show you an example of the first style here; the second style is for object-oriented modules and is described in the next chapter. (Object-oriented modules should export nothing, since the whole idea of methods is that Perl finds them for you automatically, based on the type of the object.)

To construct a module called Bestiary, create a file called *Bestiary.pm* that looks like this:

```
package    Bestiary;
require    Exporter;
our @ISA       = qw(Exporter);
our @EXPORT    = qw(camel);   # Symbols to be exported by default
our @EXPORT_OK = qw($weight);  # Symbols to be exported on request
our $VERSION   = 1.00;        # Version number
### Include your variables and functions here
sub camel { print "One-hump dromedary" }
$weight = 1024;1;
```

A program can now say use Bestiary to be able to access the camel function (but not the $weight variable), and use Bestiary qw(camel $weight) to access both the function and the variable.

c) The Exporter Module

The **Exporter** module supplies the **import** function required by the **use** statement to import functions. Export allows to export the functions and variables of modules to user's namespace using the standard import method. This way, we don't need to create the objects for the modules to access it's members.

@EXPORT and @EXPORT_OK are the two main variables used during export operation.

@EXPORT contains list of symbols (subroutines and variables) of the module to be exported into the caller namespace.

@EXPORT_OK does export of symbols on demand basis.

Example:

Let us use the following sample program to understand Perl exporter.

```
package Arithmetic;
use Exporter;
# base class of this(Arithmetic) module
@ISA = qw(Exporter);
# Exporting the add and subtract routine
@EXPORT = qw(add subtract);
# Exporting the multiply and divide  routine on demand basis.
@EXPORT_OK = qw(multiply divide);

sub add
{
```

```
    my ($no1,$no2) = @_;
    my $result;
    $result = $no1+$no2;
    return $result;
}
```

d) **Comparing use and require:**

When you import a module, you can use one of two keywords: **use** or **require.**

Use :

1. The method is used only for the modules(only to include .pm type file)
2. The included objects are varified at the time of compilation.
3. No Need to give file extension.

Syntax:

```
use Module;
and
use Module LIST;
```

Example:

```
use MyMathLib qw/add square/;
```

Require:

1. The method is used for both libraries and modules.
2. The included objects are varified at the run time.
3. Need to give file Extension.

Syntax:

```
require Module;
```

Example:

```
require 'Fcntl.pl';
require 5.003;
```


- **no:**

If MODULE supports it, then **no** calls the **unimport** function defined in MODULE to unimport all symbols from the current package, or only the symbols referred to by LIST.

It can be said that **no** is opposite of **import.**

Return Value

Nothing

Syntax

```
no Module VERSION LIST
no Module VERSION
no MODULE LIST
no MODULE
```

- **do:**

When supplied a block, do executes as if BLOCK were a function, returning the value of the last statement evaluated in the block.

When supplied with EXPR, do executes the file specified by EXPR as if it were another Perl script.

If supplied a subroutine, SUB, do executes the subroutine using LIST as the arguments, raising an exception if SUB hasn.t been defined

Syntax

do BLOCK

do EXPR

do SUB(LIST)

Example

Following are the usage...

eval `cat stat.pl`;

is equivalent to

do 'stat.pl';

e) Scope:

Scope refers to the visibility of variables. In other words, which parts of your program can see or use it. Normally, every variable has a global scope. Once defined, every part of your program can access a variable.

It is very useful to be able to limit a variable's scope to a single function. In other words, the variable wil have a limited scope. This way, changes inside the function can't affect the main program in unexpected ways, func7.pl

```
firstSub("AAAAA", "BBBBB");
sub firstSub{
   local ($firstVar) = $_[0];
   my($secondVar) = $_[1];
   print("firstSub: firstVar = $firstVar\n");
   print("firstSub: secondVar = $secondVar\n\n");
   secondSub();
   print("firstSub: firstVar = $firstVar\n");
   print("firstSub: secondVar = $secondVar\n\n");
}
sub secondSub{
   print("secondSub: firstVar = $firstVar\n");
   print("secondSub: secondVar = $secondVar\n\n");
   $firstVar = "ccccC";
   $secondVar = "DDDDD"; print("secondSub:
   firstVar = $firstVar\n"); print("secondSub:
   secondVar = $secondVar\n\n");
```

}

Working with Files:

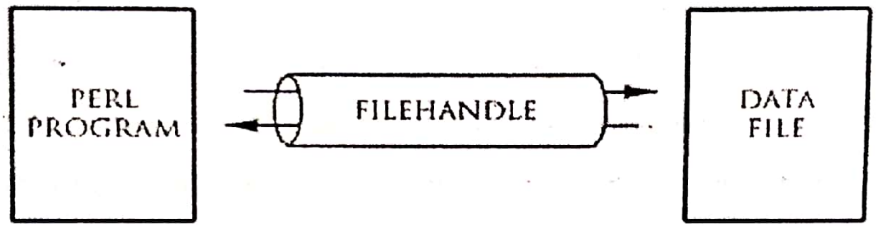
Filehandles:

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**.

The general scheme looks like this:



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 open function and EXPR is the expression having file name and mode of opening the file.

Following is the syntax to open file.txt in read-only mode. Here less than <signe 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");
while(<DATA>)
{
print "$_";
```

}

Open Function

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

open(DATA, ">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"); To truncate the file first:

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

You can open a file in 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 possible values of different modes

Entities	Definition
< or r	Read Only Access
> or w	Creates, Writes, and Truncates
>> or a	Writes, Appends, and Creates
+< or r+	Reads and Writes
+> or w+	Reads, Writes, Creates, and Truncates
+>> 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 runcate 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 possible values of MODE

Value	Definition
O_RDWR	Read and Write
O_RDONLY	Read Only
O_WRONLY	Write Only
O_CREAT	Create the file
O_APPEND	Append the file
O_TRUNC	Truncate the file
O_EXCL	Stops if file already exists
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 File handles :

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 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 rename takes two arguments and it just rename existing file

Deleting an exiting file

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

```
#!/usr/bin/perl
unlink ("/usr/test/file1.txt");
```

Locating Your Position Within 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;
```

Data Manipulation:

Perl was originally designed as a system for processing logs and reporting on the information.The data-manipulation features built into Perl, from the basics of numerical calculations through to basic string handling.

a) Working with Numbers

Numbers are scalar data. They exist in PERL as real numbers, float, integers, exponents,

b) abs-the Absolute Value

Returns the absolute value of its argument. If pure interger value is passed then it will return it as it is but if a string is passed then it will return zero. If VALUE is omitted then it uses $_.

Return Value:

Returns the absolute value of its argument.

Syntax:

```
abs VALUE
abs
```

Example:

print abs(-1.295476);

This should print a value of 1.295476. Supplying a positive value to abs will return the same positive value or, more correctly, it will return the nondesignated value: all positive values imply a + sign in front of them.

c) int-Converting Floating Points to Integers

Returns the integer element of EXPR, or $_ if omitted. The int function does not do rounding. If you need to round a value up to an integer, you should use sprintf.

Return Value:

Integer part of EXPR.

Example:

printint abs(-1.295476);

This should print a value of 1. The only problem with the int function is that it strictly removes the fractional component of a number; no rounding of any sort is done. If you want to return a number that has been rounded to a number of decimal places, use the printf or sprintf function:

printf("%.2f",abs(-1.295476));

This will round the number to two decimal places-a value of 1.30 in this example.

Note that the 0 is appended in the output to show the two decimal places.

d) exp-Raising e to the Power:

To perform a normal exponentiation operation on a number, you use the ** operator:

$square - 4**2;

This returns 16, or 4 raised to the power of 2. If you want to raise the natural base number e to the power, you need to use the exp function:

exp EXPR

exp

If you do not supply an EXPR argument, exp uses the value of the $_variable as the exponent. For example, to find the square of e:

$square -exp(2);

e) sqrt-the Square Root

To get the square root of a number, use the built-in sqrt function:

$var-sqrt(16384);

To calculate the n^{th} root of a number, use the ** operator with a fractional number. For example, the following line

$var- 16384**(1/2); is identical to $var-sqrt(16384);

To find the cube root of 16,777,216, you might use

$var- 16777216**(1/3);

which should return a value of 256.

Working with Strings

String Concatenation

Method 1 - using Perl's dot operator

Concatenating strings in Perl is very easy. There are at least two ways to do it easily. One way to do it - the way I normally do it - is to use the "dot" operator (i.e., the decimal character).

The simple example shown below demonstrates how you can use the dot operator to merge, or concatenate, strings. For simplicity in this example, assume that the variable $name is assigned the value "checkbook". Next, you want to pre-pend the directory "/tmp" to the beginning of the filename, and the filename extension ".tmp" to the end of $name. Here's the code that creates the variable $filename:

```
$name = "checkbook";
$filename = "/tmp/" . $name . ".tmp";
# $filename now contains "/tmp/checkbook.tmp"
```

Method 2 - using Perl's join function : A second way to create the desired variable $filename is to use the join function. With the join function, you can merge as many strings together as you like, and you can specify what token you want to use to separate each string.

The code shown in Listing 2 below uses the Perl join function to achieve the same result as the code shown in Listing 1:

```
$name = "checkbook";
$filename = join "", "/tmp/", $name, ".tmp";
# $filename now contains "/tmp/checkbook.tmp
```

String Length

To get the length of a string in Perl, use the Perl length function on your string, like this:

```
$size = length $last_name;
```

The variable $size will now contain the string length, i.e., the number of characters in the variable named $last_name.

Example

Here's a more complete script to demonstrate this Perl string length technique:

```
$a = "foo, bar, and baz";

print length $a;
```

When that little script is run through the Perl interpreter it will print the number 17, which is the length of that string (i.e., the number of characters in the Perl string).

Case Modifications

The four basic functions are lc, uc, lcfirst, and ucfirst. They convert a stringto all lowercase, all uppercase, or only the first character of the string to lowercase oruppercase, respectively.

For Example:

```
$string = "The Cat Sat on the Mat"; printlc($string) #
Outputs 'the cat sat on the mat' printlcfirst($string) #
Outputs 'the Cat Sat on the Mat' printuc($string) # Outputs
```

'THE CAT SAT ON THE MAT' printucfirst($string) #

Outputs 'The Cat Sat on the Mat'

These functions can be useful for "normalizing" a string into an all uppercase or lowercase format—useful when combining and de-duping lists when using hashes.

- **End-of-Line Character Removal**

The chop function can be used to strip the last character off any expression

```
while(<FH>)
{
    chop;..
}
```

- **String Location**

The index() function is used to determine the position of a letter or a substring in a string. For example, in the word "frog" the letter "f" is in position 0, the "r" in position 1, the "o" in 2 and the "g" in 3. The substring "ro" is in position 1. Depending on what you are trying to achieve, the index() function may be faster or more easy to understand than using a regular expression or the split() function.

Example:

```
#!/usr/bin/perl
use strict;
use warnings;
my $string = 'perlmeme.org';
my $char = 'l';
my $result = index($string, $char);
print "Result: $result\n";
```

Instead of looping through every occurrence of a letter in a string to find the last one, you can use the rindex() function. This works exactly the same as index() but it starts at the end of the string. The index value returned is string from the start of the string though.

Example:

```
#!/usr/bin/perl
use strict;
use warnings;
my $string = 'perlmeme.org';
my $char = 'e';
my $result = rindex($string, $char);
print "Result: $result\n"
```

- **Extracting Substrings**

The substr() function is used to return a substring from the expression supplied as it's first argument. The function is a good illustration of some of the ways in which Perl is different from other languages you may have used. substr() has a variable number of arguments, it can be told to start at an offset from either

end of the expression, you can supply a replacement string so that it replaces part of the expression as well as returning it, and it can be assigned to!.

Example:

```
#!/usr/bin/perl
use strict;
use warnings;
my $string = 'Now is the time for all good people to come to the aid of their party';
my $fragment =  substr $string, 4;
print "  string: <$string>\n";
print "fragment: <$fragment>\n";
```