Birzeit University - Faculty of Engineering & Technology
Electrical & Computer Engineering Department - ENCS313
Linux laboratory

Experiment #10
Learning & Programming Perl
Part I

0.1 Introduction

Perl (Practical Extraction and Reporting Language) was developed by Larry Wall in 1987 as a general-purpose Unix scripting language to make report processing easier. Since then, it has undergone many changes and revisions. The latest major stable revision of Perl 5 is 5.18, released in May 2013[1].

Perl is designed to assist the programmer with common tasks that are probably too heavy or too portability-sensitive for the shell, and yet too weird or short-lived or complicated to code in C or some other UNIX glue language[2][3].[4]. The Perl language borrows features from other programming languages including C, shell scripting (sh), AWK, and sed. They provide powerful text processing facilities without the arbitrary data-length limits of many contemporary Unix tools, facilitating easy manipulation of text files. Perl 5 gained widespread popularity in the late 1990s as a CGI scripting language, in part due to its parsing abilities.

The current experiment intends to present to students the perl programming language. First the language syntax will be presented: Scalar Data, Arrays and List Data, Control Structures, Hashes, Basic I/O, Regular Expressions (like the ones we've seen with shell scripting). In addition, students will be shown how to build perl functions, how to do File and Directory Manipulation, Process Management and how to build packages and modules (and much more).

Since all the above material seems too much to fit in one single experiment, the content on perl will be split into 2 experiments. In the first part, we'll go over learning how to program in perl. In the second part, you'll get your hands wet in programming in perl where we'll see some advanced topics.

It is worth mentioning that perl comes pre-installed with all Linux and Unix distributions. Perl packages can also be installed on windows platforms. The most famous perl distributions for windows are ActivePerl and Strawberry Perl.

0.2 Objectives

The objectives of the experiment is to learn the following:

- Perl syntax and control structures.
- Show some examples about perl scripts and get some hands on regarding perl programming.
- Show some Perl functions.
- Show text processing using Perl.

[1] Perl: From Wikipedia, the free encyclopedia
[2] Randal L. Schwartz, Tom Christiansen and Larry Wall - Learning Perl, 2nd Edition
[3] Larry Wall, Tom Christiansen & Randal L. Schwartz - Programming Perl - 2nd Edition
[4] Tom Christiansen & Nathan Torkington - Perl Cookbook - 1st Edition

- Show how to debug Perl scripts with Perl debugger.
- Show how to create formats with Perl.
- Explain What are packages in Perl.
- Show how to build Perl modules.

0.3 Basic Concepts

Perl is an interpreted language. Perl interpreter completely parses and compiles the program into an internal format before executing any of it. This means that you can never get a syntax error from the program once the program has started, and that the whitespace and comments simply disappear and won't slow the program down. This compilation phase ensures the rapid execution of Perl operations once it is started, and it provides additional motivation for dropping C as a systems utility language merely on the grounds that C is compiled.

So Perl is like a compiler and an interpreter. It's a compiler because the program is completely read and parsed before the first statement is executed. It's an interpreter because there is no object code sitting around filling up disk space.

A final comment before we go deep into Perl. To know which version of Perl is installed on your machine, execute the following command on your shell:

```
perl -v
```

You will probably get an output similar to the below statements:

```
This is perl, v5.8.0 built for i386-linux-thread-multi
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2002, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using 'man perl' or 'perldoc perl'.  If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

0.3.1 Structure of a perl program

Perl program is a text file. You can use any text editor to create the program such as `gedit`, `emacs` or even `vi`. Normally the following line will be the first line.

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

This tells Linux to use `/usr/bin/perl` executable to interpret rest of the lines in the program. This line may vary depending on the location of perl binary. Sometimes it may be `/usr/local/bin/perl` or some other place.

Commonly `.pl` extension is used (e.g. `script.pl`), however you can write the perl script without extension also. It will still work fine [5].

[5] An introduction to perl language BE CSE students - Raman (http://ramanchennai.wordpress.com/2010/11/21/an-introduction-to-perl-language-be-cse-students/)

0.3.2 Comments

The symbol `#` is used for comments. All text from `#` till end of line is treated as comment.

Example

```
# This is a full line comment
```

Note: There is no multiline comment as in the C-language.

0.3.3 Statement terminator

All Statements end with `;` like C-language.

0.4 Scalar Data

A scalar is the simplest kind of data that Perl manipulates. A scalar is either a number (like 4 or 3.25e20) or a string of characters (like hello). Although you may think of numbers and strings as very different things, Perl uses them nearly interchangeably.

0.4.1 Numbers

Perl supports decimal, octal and hexadecimal (hex) numbers. Octal numbers start with a leading `0`, and hex numbers start with a leading `0x`

Example

```
0377 # 377 octal, same as 255 decimal
0xFF # same as 255 decimal
```

0.4.2 Strings

Strings are sequences of characters. Strings can be single-quoted strings or double-quoted strings.

Single-Quoted Strings

A single-quoted string is a sequence of characters enclosed in single quotes.

Example

```
'hello' # five characters:  h, e, l, l, o
'don\'t' # five characters:  d, o, n, single-quote, t
```

Note: The \n within a single-quoted string is not interpreted as a newline, but as the two characters backslash and n.

Double-Quoted Strings

A double-quoted string acts a lot like a C string. Once again, it's a sequence of characters, although this time enclosed in double quotes. But now the backslash takes on its full power to specify certain control characters.

Example

```
"hello world\n" # hello world, and a newline
"coke\tsprite" # a coke, a tab, and a sprite
```

Note: Variables are interpolated with double-quoted strings, meaning that scalar and array variables within the strings are replaced with their current values when the strings are used. Check the below examples.

0.5 Scalar Operators

Perl's operators and expressions are similar to the operators provided by the C-language. For numbers, the operators are `+`, `-`, `*`, `/`, `**` (exponentiation) and `%` (modulus).

For strings, there is an operator concatenation with the "." operator.

Example

```
"hello" .  "world"          # same as "helloworld"
'hello world' .  "\n"       # same as "hello world\n"
"fred" .  " " .  "barney"   # same as "fred barney"
```

Below are the main numeric and string comparison operators:

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

Note that the numeric and string comparisons in perl are roughly *opposite* of what they are for the shell scripting seen in the previous experiments.

0.6 Scalar Variables

Scalar variables can contain numeric or string values. The name of the variable should be prefixed with $ symbol.

Examples

```
$name = "BZU - encs311"          # String context
$age = 30                        # Numeric context
$age = $age + 1                  # Treated as numeric
$str = $name . " " . "Linux lab" # Treated as string
```

Note:

You can use the operators in pretty much the same way you have been using them in the C-language. Below are simple examples to put things in order:

```
$a = 10;
$a = $a + 1;    # Same as $a++;

$a = $a * 3;    # Same as $a *= 3;

$a = $a - 1;    # Same as $a--

$a = ($a + 1) * 4
```

0.7 The chop and chomp Functions

A useful built-in function is `chop`. This function takes a single argument within its parentheses - the name of a scalar variable - and removes the last character from the string value of that variable. For example:

```
$x = "hello world";
chop($x); # $x is now "hello worl"
```

Note that function `chop` returns the character that has been removed (in the above example, it returns "d"). Note as well that when function `chop` is given an empty string, it does nothing, and returns nothing.

If you're not sure whether the variable has a newline on the end, you can use the slightly safer `chomp`q operator, which removes only a newline character. For example:

```
$x = "hello world\n";
chomp($x); # $x is now "hello world"
chomp($x); # no change in $x
```

0.8 Some built-in case-shifting for strings

The case-shifting string escapes can be used to alter the case of letters as in the following examples:

```
$lab = "encs313";
$biglab = "\U$lab";      # $biglab = "ENCS313"
$caplab = "\u$lab";      # $caplab = "Encs313"
$lowlab = "\L$biglab";   # $lowlab = "encs313"
```

You can also use the functions: `uc`, `ucfirst`, `lc`, and `lcfirst`.

0.9 Input and Output

Keyboard inputs can be accepted using `<STDIN>`.

Output to screen can be made using function `print`.

Examples

Type the following example in a file called `in_out.pl`:

```
#!/usr/bin/perl

print "Please enter your name:";
$name = <STDIN>;

print "Welcome $name\n";
```

Run the perl script as follows:

`perl in_out.pl`

and check on the results you get.

0.10 Arrays and List Data

A `list` is ordered scalar data. An array is a variable that holds a list.

Examples

```
(1,2,3)                    # array of three values 1, 2, and 3
("fred",4.5)               # two values, "fred" and 4.5
()                         # the empty list (zero elements)

(1 .. 5)                   # same as (1, 2, 3, 4, 5)
(1.3 .. 6.1)               # same as (1.3,2.3,3.3,4.3,5.3)

@a = ("great","encs313"); # a is an array that contains "great" and "encs313"
```

A better way to define an array is to use the "quote word" function, which creates a list from the nonwhitespace parts between the parentheses. The above array can be thus written as:

```
@a = qw(great encs313);
```

Note

To print out the content of the array a, use the following:

```
print("The content of array a is ",@a,"\n");
```

0.10.1 Operations on arrays

Below are some operations on arrays:

```
@fred = (1,2,3); # The fred array gets a three-element literal
@barney = @fred; # now that is copied to @barney

@fred = qw(one two);
@barney = (4,5,@fred,6,7); # @barney becomes
                           # (4,5,"one","two",6,7)

@barney = (8,@barney);     # puts 8 in front of @barney
@barney = (@barney,"last");# @barney is now (8,4,5,"one","two",6,7,"last")

($a,$b,$c) = (1,2,3);      # give 1 to $a, 2 to $b, 3 to $c
($a,$b) = ($b,$a);         # swap $a and $b
($d,@fred) = ($a,$b,$c);   # give $a to $d, and ($b,$c) to @fred
($e,@fred) = @fred;        # remove first element of @fred to $e
                           # this makes @fred = ($c) and $e = $b

@fred = (4,5,6);           # initialize @fred
$a = @fred;                # $a gets 3, the current length of @fred
                           # same as variable $#fred

($a) = @fred;              # $a gets the first element of @fred

$b = $fred[0];             # give 4 to $b (first element of @fred)
```

0.10.2 Useful functions for arrays

Below is list of useful functions that can be used in conjunction with arrays:

The push and pop functions

One common use of an array is as a stack of information, where new values are added to and removed from the right-hand side of the list. Below are some examples:

```
@mylist = (1,2,3);
push(@mylist,4,5,6);        # @mylist = (1,2,3,4,5,6)

$newValue = 7
push(@mylist,$newvalue);    # like @mylist = (@mylist,$newvalue)
                            # mylist = (1,2,3,4,5,6,7)

$oldValue = pop(@mylist);   # removes the last element of @mylist
                            # oldValue = 7
```

Note that the `pop` function returns `undef` if given an empty list.

The shift and unshift functions

The `push` and `pop` functions do things to the *right* side of a list (the portion with the highest subscripts). Similarly, the `unshift` and `shift` functions perform the corresponding actions on the *left* side of a list. Below are some examples:

```
unshift(@mylist,$a);       # like @mylist = ($a,@mylist);
unshift(@mylist,$a,$b,$c); # like @mylist = ($a,$b,$c,@mylist);
$x = shift(@mylist);       # like ($x,@mylist) = @mylist;

@mylist = (5,6,7);
unshift(@mylist,2,3,4);    # @mylist is now (2,3,4,5,6,7)
$x = shift(@mylist);       # $x gets 2, @mylist is now (3,4,5,6,7)
```

As with `pop`, `shift` returns `undef` if given an empty array variable.

The reverse function

The reverse function reverses the order of the elements of its argument, returning the resulting list. Below are some examples:

```
@a = (7,8,9);
@b = reverse(@a);    # gives @b the value of (9,8,7)
@b = reverse(7,8,9); # same thing
```

The sort function

The `sort` function takes its arguments, and sorts them as if they were single strings in ascending ASCII order. Below are some examples:

```
@x = sort("small","medium","large"); # @x gets "large","medium","small"

@y = (1,2,4,8,16,32,64);
@y = sort(@y);                       # @y gets 1,16,2,32,4,64,8
```

0.11 Function Reference

The functions that we've seen above are a subset of the built-in Perl functions. Since describing each function in detail and providing an example of its usage can be exhaustive, we'll simply provide the list of Perl built-in functions sliced by category. Students are invited to research these functions and their usage them on need-basis.

You'll notice in particular that many of Perl functions have similar names and behavior to their counterparts in the C-language.

0.11.1 Perl Functions by Category

Here are Perl's functions and function-like keywords, arranged by category. Note that some functions appear under more than one heading[6].

Scalar manipulation

`chomp, chop, chr, crypt, hex, index, lc, lcfirst, length, oct, ord, pack, q//, qq//, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///`

Regular expressions and pattern matching

`m//, pos, qr//, quotemeta, s///, split, study`

Numeric functions

`abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand`

Array processing

`pop, push, shift, splice, unshift`

List processing

`grep, join, map, qw//, reverse, sort, unpack`

Hash processing

`delete, each, exists, keys, values`

Input and output

`binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, rewinddir, seek, seekdir, select, syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write`

Fixed-length data and records

`pack, read, syscall, sysread, syswrite, unpack, vec`

Filehandles, files, and directories

`chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, stat, symlink, sysopen, umask, unlink, utime`

Flow of program control

`caller, continue, die, do, dump, eval, exit, goto, last, next, redo, return, sub, wantarray`

Scoping

`caller, import, local, my, package, use`

Miscellaneous

`defined, dump, eval, formline, local, my, prototype, reset, scalar, undef, wantarray`

[6]Ellen Siever, Stephen Spainhour & Nathan Patwardhan - Perl in a Nutshell - 1st Edition

Processes and process groups

`alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority sleep, system, times, wait, waitpid`

Library modules

`do, import, no, package, require, use`

Classes and objects

`bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use`

Low-level socket access

`accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair`

System V interprocess communication

`msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite`

Fetching user and group information

`endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent`

Fetching network information

`endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent`

Time

`gmtime, localtime, time, times`

0.12 Control Structures

Like other programming languages, Perl provides control structures to steer the execution of the Perl scripts.

0.12.1 if/elsif/else/unless statements

The syntax of `if` statement is:

```
if ( condition) {

}

elsif (condition){

}

else {

}
```

The `if` statement is similar to the `if` in the C-language, except:

- The curly braces are needed even for single statements following an `if` statement.
- The `else if` in the C-language is replaced by `elsif`.

Example

Type the following example in a file called `test_if.pl`:

```
#!/usr/bin/perl

print "Please enter your grade:";
$grade=<STDIN>;

chomp($grade);

if ($grade > 79){
   print "passed with distinction\n";
}
elsif ($grade < 60){
   print "failed\n";
}
else {
   print "passed\n";
}
```

Run the perl script as follows:

```
perl test_if.pl
```

and check on the results you get.

Sometimes, it is more natural to say *do that if this is false* rather than *do that if not this is true*. Perl handles this with the `unless` variation.

Example

Type the following example in a file called `test_unless.pl`:

```
#!/usr/bin/perl

print "Please enter your grade:";
$grade=<STDIN>;

chomp($grade);

unless ($grade < 79) {
    print "passed with distinction\n";
}
```

Run the perl script as follows:

```
perl test_unless.pl
```

and check on the results you get.

0.12.2 while/until statement

Perl can iterate using the `while` statement. the syntax and usage is similar to the `while` statement used in the C-language.

Example

Type the following example in a file called `test_while.pl`:

```
#!/usr/bin/perl

$i = 0;

while ($i < 10) {
   print "i = $i\n";

   $i++;
}
```

Run the perl script as follows:

`perl test_while.pl`

and check on the results you get.

Sometimes it is easier to say *until something is true* rather than *while not this is true*. Once again, Perl has the answer. Replacing the `while` with `until` yields the desired effect. The usage of the `until` statement is similar to its usage in the C-language.

You can also use the `do{} while/until` statement in a similar way as you used it for the C-language. Check the below example.

Example

Type the following example in a file called `test_do_until.pl`:

```
#!/usr/bin/perl

$stops = 0;
do {
    $stops++;

    print "Next stop? ";
    chomp($location = <STDIN>);

} until $stops > 5 || $location eq 'home';
```

Run the perl script as follows:

`perl test_do_until.pl`

and check on the results you get.

0.12.3 for statement

The `for` statement behaves in the same way as you're used to in the C-language. Below is an example on how to use it.

Example

Type the following example in a file called `test_for.pl`:

```
#!/usr/bin/perl

for ($i = 1; $i <= 10; $i++) {
    print "$i ";
}
```

Run the perl script as follows:

`perl test_for.pl`

and check on the results you get.

0.12.4 foreach statement

The `foreach` statement takes a list of values and assigns them one at a time to a scalar variable, executing a block of code with each successive assignment. It looks like this:

```
foreach $i (@some_list) {
    statement_1;
    statement_2;
    statement_3;
}
```

Example

Type the following example in a file called `test_foreach.pl`:

```
#!/usr/bin/perl

@a = (1,2,3,4,5);
foreach $b (reverse @a) {
    print $b;
}
```

Run the perl script as follows:

```
perl test_foreach.pl
```

and check on the results you get.

0.12.5 Default scalar variable $_

`$_` is called default variable. It will be used if no other variable is specified.

Example

Type the following example in a file called `test_foreach_1.pl`:

```
#!/usr/bin/perl

@a = (1,2,3,4,5);
foreach (reverse @a) {
    print;
}
```

0.13 Hashes

A *hash* is like the array that we discussed earlier, in that it is a collection of scalar data, with individual elements selected by some index value. Unlike a list array, the index values of a hash are not small nonnegative integers, but instead are arbitrary scalars. These scalars (called *keys*) are used later to retrieve the values from the array. Hashes are called *associative arrays*.

Hashes are thus key-value pairs. Hash variables will have `%` as prefix. The contents of hash are called *values* and index is called `key`. Below is an example of a hash.

Example

```
%fruits = (
'apple' =>'red',
'banana'=>'yellow',
'grape' =>'black'
);
```

The hash can be populated also as follows:

```
%fruits = ('apple,'red','banana','yellow','grape','black');
```

Individual elements of hash are accessed by means of `$hash{key}` as follows:

```
print "colour of apple is $fruits{apple}\n";
```

Below is a full example.

Example

Type the following example in a file called `test_hash.pl`:

```
#!/usr/bin/perl

%fruits = (
'apple' =>'red',
'banana'=>'yellow',
'grape' =>'black'
);

foreach $f (keys %fruits) {
   print "Color of $f is $fruits{$f}\n";

$lab{"linux"} = "encs313";  # create key "linux", values "encs313"

print $lab{"linux"};        # prints "encs313"
}
```

Run the perl script as follows:

`perl test_hash.pl`

and check on the results you get.

Hashes have some functions that let you access their fields. Below are some of these functions:

0.13.1 The `keys` function

The `keys(%hashname)` function yields a list of all the current keys in the hash `%hashname`. The parentheses are optional (e.g. you can use it as `keys %hashname`).

If there are no elements to the hash, then `keys` returns an empty list.

Example

Type the following example in a file called `test_hash_keys.pl`:

```
#!/usr/bin/perl

$lab{"linux"} = "encs313";
$lab{"rt"} = "encs514";

@list = keys(%lab); # @list gets ("linux", "rt") or ("rt", "linux")

foreach $key (keys (%lab)) { # once for each key of %lab
    print "at $key we have $lab{$key}\n"; # show key and value
}
```

Run the perl script as follows:

`perl test_hash_keys.pl`

and check on the results you get.

0.13.2 The `values` function

The `values(%hashname)` function returns a list of all the current values of the `%hashname`. The parentheses are optional (e.g. you can use it as `values %hashname`).

Example

Type the following example in a file called `test_hash_values.pl`:

```
#!/usr/bin/perl

$lab{"linux"} = "encs313";
$lab{"rt"} = "encs514";

@list = values(%lab); # @list gets ("encs313", "encs514")

print "The values are: ", @list, "\n";
```

Run the perl script as follows:

`perl test_hash_values.pl`

and check on the results you get.

0.13.3 The `each` function

The function `each %hashname` returns a key-value pair as a two-element list. On each evaluation of this function for the same hash, the next successive key-value pair is returned until all the elements have been accessed. When there are no more pairs, `each` returns an empty list.

Example

Type the following example in a file called `test_hash_each.pl`:

```
#!/usr/bin/perl

%lastname =(
'Mohammad' => 'Khawaja',
'Nabeel'   => 'Isleem',
'Fadia'    => 'Asmar',
'Ayman'    => 'Khalaf'
);

while (($first,$last) = each(%lastname)) {
    print "The last name of $first is $last\n";
}
```

Run the perl script as follows:

`perl test_hash_each.pl`

and check on the results you get.

0.13.4 The `delete` function

Perl provides the `delete` function to remove hash elements. It removes the key-value pair from the hash.

Example

Type the following example in a file called `test_hash_delete.pl`:

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

```
%lastname =(
'Mohammad' => 'Khawaja',
'Nabeel'   => 'Isleem',
'Fadia'    => 'Asmar',
'Ayman'    => 'Khalaf'
);

delete $lastname{"Nabeel"};

while (($first,$last) = each(%lastname)) {
    print "The last name of $first is $last\n";
}
```

Run the perl script as follows:

`perl test_hash_delete.pl`

and check on the results you get.

0.14 Subroutines (Functions)

A user-define function, more commonly called *subroutines* can be defined using sub keyword. The arguments passed will be in a default array `@_`.

Within the subroutine body, you may access or give values to variables that are shared with the rest of the program (*global variable*).

Example

```
sub say_what {
  print "hello, $what\n";
}
```

In the above example, the variable `$what` refers to a global variable shared with the rest of the program.

If you wish to pass arguments to a subroutine, you can use the default array `@_` as mentioned earlier.

Example

Type the following example in a file called `test_sub_1.pl`:

```
#!/usr/bin/perl

$v1 = 10; $v2 = 20;

add($v1,$v2);

sub add
{
  ($a,$b) = @_;

  print $a + $b, "\n";
}
```

Run the perl script as follows:

`perl test_sub_1.pl`

and check on the results you get.

You can return value using the `return` statement that you are used to from the C-language.

0.14.1 Scope of variables

By default all variables are global, that is available throughout the program. You can limit scope to a block or a subroutine by using the statement `my`. Check the below example.

Example

Type the following example in a file called `test_sub_2.pl`:

```
#!/usr/bin/perl

$v1 = 10; $v2 = 30; $v3 = 30; #v1, v2, v3 global variables

$v3 = add($v1,$v2);

sub add{
  my ($i,$j) = @_;

  print "inside add sub value of i = $i j = $j\n";

  print "inside add sub value of globals v1 = $v1 v2 = $v2 v3 = $v3\n";

  return $i + $j;
}
print "Value of globals v1 = $v1 v2 = $v2\n";

print "Value of scoped variables v3 = $v3\n";

print "Value of variables inside sub i = $i j = $j\n";
```

Run the perl script as follows:

```
perl test_sub_2.pl
```

and check on the results you get.

Below is a little bit more complicated example on using arguments.

Example

Type the following example in a file called `test_sub_3.pl`:

```
#!/usr/bin/perl

sub add {
    $sum = 0;         # initialize the sum
    foreach $_ (@_) {
        $sum += $_;   # add each element
    }
return  $sum;         # last expression evaluated: sum of all elements
}
print add(1,2,3,4,5), "\n";     # prints 15
print add(1,2,3,4,5,6,7), "\n"; # prints 28
print add(1..5), "\n";          # also prints 15, because 1..5 is expanded
```

Run the perl script as follows:

```
perl test_sub_3.pl
```

and check on the results you get. Note the difference between the two `add` subroutines in the current example and in the previous one.

Perl gives you a second way to create *private* variables, using the `local` function. Check the

below example.

Example

Type the following example in a file called `test_local.pl`:

```
#!/usr/bin/perl

$value = "original";

tellme();
spoof();
tellme();

sub spoof {
    local ($value) = "temporary";
    tellme();
}

sub tellme {
    print "Current value is $value\n";
}
```

Run the perl script as follows:

```
perl test_local.pl
```

and check on the results you get.

0.14.2 using the `strict` pragma

In perl you need not define variables before using them since by default all variables are global. However, this may lead to errors due scope conflict or errors in naming. Using the `strict` pragma helps in avoiding that problem. Check the below examples.

Example

Type the following example in a file called `test_strict_error.pl`:

```
#!/usr/bin/perl

use strict;

$v1 = 10; $v2 = 20;

add($v1,$v2);

sub add {

   ($a,$b) = @_;

   print $a + $b, "\n";
}
```

Run the perl script as follows:

```
perl test_strict_error.pl
```

and check on the results you get. You'll notice that perl will complain and will exit immediately. The correct version that will run is shown below:

Example

Type the following example in a file called `test_strict_ok.pl`:

```
#!/usr/bin/perl

use strict;

my $v1 = 10;
my $v2 = 20;

add($v1,$v2);

sub add {

   ($a,$b) = @_;

   print $a + $b, "\n";
}
```

Run the perl script as follows:

```
perl test_strict_ok.pl
```

and check on the results you get.

0.15 Using constants

Like C-language which offers the `#define` pragma to define constants, Perl offers a similar functionality to define constants by using the pragma `constant` as follows:

```
use constant WHITE => 1;
use constant BLACK => 0;
```

Constants can be grouped in one block using the pragma `constant` as follows:

```
use constant {
  SEC   => 0,
  MIN   => 1,
  HOUR  => 2,
  MDAY  => 3,
  MON   => 4,
  YEAR  => 5,
  WDAY  => 6,
  YDAY  => 7,
  ISDST => 8,
};
```

You can even define an array of constants as follows:

```
use constant WEEKDAYS => qw(
  Sunday Monday Tuesday Wednesday Thursday Friday Saturday
);
```

To print the value of a constant, check the below example.

Example

Type the following example in a file called `test_constant_1.pl`:

```
#!/usr/bin/perl

use constant WEEKDAYS =>
    [ qw{ Sunday Monday Tuesday Wednesday Thursday Friday Saturday } ];

my $i = 0;

foreach (@{+WEEKDAYS}) {
    print "WeekDay[$i] = ", WEEKDAYS->[$i], "\n";

    $i = $i + 1;
}
```

Run the perl script as follows:

`perl test_constant_1.pl`

and check on the results you get.

0.16 File handling

A *filehandle* in a Perl program is the name for an I/O connection between your Perl process and the outside world. Perl provides `<STDIN>` for standard input, `<STDOUT>` for standard output and `<STDERR>` for standard error output. These names are the same as those used by the C and C++ "standard I/O" library package.

Users are advised to have their file handles names in uppercase like Perl's convention.

To open a file handle for reading, use the `open` function as follows:

```
open(FILEHANDLE,"somename");
```

To open a file for file for writing, do the following:

```
open(FILEHANDLE,">outFile");
```

Note the greater-than sign (`>`) in front of the file `outFile`. In addition, to append to a file, you can use two greater-than signs as a prefix as follows:

```
open(FILEHANDLE,">>appendLogFile");
```

When you are finished with a filehandle, you may close it with the `close` operator, like as follows:

```
close(FILEHANDLE);
```

Check the folowing example that relates to file handles.

Example

Type the following example in a file called `test_file_1.pl`:

```
#!/usr/bin/perl

open($FH_READ, "data.txt");   # open file read only

open($FH_WRITE,">udata.txt"); # Open file write mode

while ($line = <$FH_READ> ) { # cread one line from file
   print "line = $line";      # cdisplay content on screen
```

```
   print $FH_WRITE uc($line); # Write upper cased content to new file
}
close($FH_READ);
close($FH_WRITE);
```

Create a file called `data.txt` and type down the following data:

```
Mohammad Khawaja
Nabeel Isleem
Fadia Asmar,
Ayman Khalaf
```

Run the perl script as follows:

```
perl test_file_1.pl
```

and check on the results you get. Open the file `udata.txt` and check on the content. Explain the results that you get.

0.16.1 How to `die` in Perl

Perl never issues a warning if a file handle is not opened successfully either for reading or for writing. Typically you'll want to check the result of the `open` and report an error if the result is not what you expect. One way of doing that is as follows:

```
unless (open (FH_WRITE,">udata.txt")) {
    print "Sorry, I couldn't open the file udata.txt for writing\n";
} else {
    # the rest of your program
}
```

A more elegant way of doing it is by using the function `die`. It acts like the function `print` by printing out a message on the standard error output and then ends the Perl process with a nonzero exit status (generally indicating that something unusual happened). You can do that as follows:

```
unless (open (FH_WRITE,">udata.txt")) {
    die "Sorry, I couldn't open the file udata.txt for writing\n";
}
```

In the above code, if the Perl process is unable to write to file `udata.txt`, it will exit after writing the error message to the screen.

An even more elegant way of doing the same thing would be as follows:

```
open (FH_WRITE,">udata.txt") ||
   die "Sorry, I couldn't open the file udata.txt for writing\n";
}
```

Where `||` is the OR operator. Below is another example on using file handles.

Example

Type the following example in a file called `test_file_2.pl`:

```
#!/usr/bin/perl

open (EP,"/etc/passwd");

while (<EP>) {
    chomp;
    print "I saw $_ in the password file!\n";
}
close(EP) || die "can't close file /etc/passwd";
```

Run the perl script as follows:

`perl test_file_2.pl`

and check on the results you get.

The function `warn` has a similar behavior to function `die` but doesn't try to exit or throw an exception. For example, you can use it as in the below incomplete piece of code:

Example

```
..
warn "Debug enabled" if $debug;
..
```

0.16.2 Checking on files

Since you might destroy files if you write to them before checking if they do exist before, Perl provides a set of options that you can use such as `-e` or `-x`. Check the below example:

Example

Type the following example in a file called `test_file_options.pl`:

```
#!/usr/bin/perl

$file = "data.txt";

if (-e $file) {
    print "The file $file exists\n";
} else {
    print "The file $file does NOT exist\n";
}
```

Run the perl script as follows:

`perl test_file_options.pl`

and check on the results you get.

Many more options are available for usage. Below is a set of useful options that you might need in the future:

File Test	Meaning
`-e`	File or directory exists
`-r`	File or directory is readable
`-w`	File or directory is writable
`-x`	File or directory is executable
`-s`	File or directory exists and has nonzero size (the value is the size in bytes)
`-z`	File exists and has zero size (directories are never empty)
`-d`	Entry is a directory
`-l`	Entry is a symbolic link
`-f`	Entry is a plain file