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

Experiment #11
Learning & Programming Perl
Part II

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 File and Directory Manipulation

0.3.1 Removing a File

Files can be removed in Perl using the function `unlink` as follows:

```
unlink data.txt
```

The above function call will remove the file `data.txt`.

The `unlink` function can take a list of names to be unlinked as in the below example:

```
#!/usr/bin/perl

unlink ("data.txt","udata.txt"); # delete the 2 files data.txt and udata.txt
unlink <*.o>;                    # just like "rm *.o" in the shell
```

0.3.2 Renaming a File

Files can be renamed in Perl using the function `rename` as follows:

```
rename(‘‘data.txt’’,’’udata.txt’’); || die ‘‘Can’t rename the file’’
```

0.3.3 Making and Removing Directories

Perl can create a directory using the `mkdir` function. It takes a name for a new directory and a mode that will affect the permissions of the created directory. Check the below example.

Example

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

```
#!/usr/bin/perl

mkdir ("newDir", 0777) || die "cannot mkdir gravelpit: $!";
```

Run the perl script as follows:

```
perl test_create_dir.pl
```

and check on the results you get. If you have the right permissions, the directory `newDir` should be created with the assigned permissions.

Directories can be removed using the `rmdir` function as follows:

```
rmdir("newDir") || die "cannot rmdir newDir:  $!";
```

0.4 Pattern Matching - Text Processing using Perl

Perl is powerful for text processing applications using a technique called regular expressions[5]. Pattern matching is more than just searching for some set of characters in your data; its a way of looking at data and processing that data in a manner that can be incredibly efficient and amazingly easy to program. You’ll see a lot of common terminology we’ve seen when we handled shell scripting in the previous experiments.

[5]http://work.lauralemay.com/samples/perl.html

0.4.1 Pattern Matching Operators and Expressions

To use pattern matching in Perl, you figure out what you want to find, you write a regular expression to find it, and then you stick that pattern in a situation where the result of finding (or not finding) that pattern makes sense. To construct patterns in this way, you use two operators: the regular expression operator `m//` and the pattern-match operator `=~`, like this:

```
if ($string =~ m/foo/) {
 # do something...
 }
```

What that test inside the `if` says is: if the string contained in $string contains the pattern `foo`, return true. Note that the operator `=~` is not an assignment operator, even though it looks like one. `=~` is used exclusively for pattern matching and is called the `binding` operator.

Usually, the letter `m` is optional and the `=~` is optional as well if the default variable `$_` is used. Check the below example:

Example

Type the following example in a file called `test_pattern_1.pl`:

```
#!/usr/bin/perl

$_= "hello how are you";

if (m/hello/){ # You can omit the letter m as explained above
  print "default variable = $_\n";

  print "found hello\n";
}
```

Run the perl script as follows:

```
perl test_pattern_1.pl
```

and check on the results you get. Notice in particular that the string `hello` was found in the default variable `$_`.

Using the `i` modifier ignores the case difference such as the below incomplete code:

```
#!/usr/bin/perl
.
.
.
if (m/hello/i){
.
.
.
}
```

In addition, you can match only if the string hello occurs at the start of the line as below:

```
.
.
.
if (m/^hello/) {
.
.
.
}
```

or occurs at the end of the line by replacing

```
if (m/^hello/) {

}
```

by

```
if (m/hello$/) {

}
```

Some tricks

The pattern `/^/` matches empty strings.
The pattern `/^$/` matches an empty line.
The pattern `/^.$/` matches lines that contain one character and one character only.
The pattern `/^..:/` matches lines that starts with two characters and a colon.

0.4.2 The `split` function

The `split` is a function that splits a string into components based on a specified delimiter, and return a list of values. Check the below example:

Example

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

```
#!/usr/bin/perl

my $line = "23/07/2013";

($day, $month, $year) = split /\//,$line;

print "day = $day, month = $month, year = $year\n";

print "length of $line is:", length $line, "\n";
```

Run the perl script as follows:
`perl test_split.pl`
and check on the results you get.

In the above example, the function `split` will use `/` as delimiter. Note that `/` is a special character. To use it right, we should use `\/` (precede `/` with a backslash).

Note as well the function `length` which returns the number of characters in a string.

0.4.3 Boundary matching

A word boundary is indicated using a `\b` escape. For example if you want to look for the string `if` when it comes alone but not in a string such as `difference` or `iffy`. So `\bif\b` will match only when the whole word `if` exists in the stringbut not when the characters `i` and `f` appear in the middle of a word. Check the below example:

Example

Type the following example in a file called `test_pattern_2.pl`:

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

```
my $str1 = "if I were king";
my $str2 = "difference";
my $str3 = "that result is iffy";

if ($str1 =~ /\bif\b/) {
  print "if has been found in str1\n";
}

if ($str2 =~ /\bif\b/) {
  print "if has been found in str2\n";
}

if ($str3 =~ /\bif/) {
  print "if has been found in str3\n";
}

if ($str1 =~ /\bif/) {
  print "if has been found in str1\n";
}

if ($str2 =~ /\bif/) {
  print "if has been found in str2\n";
}

if ($str3 =~ /\bif/) {
  print "if has been found in str3\n";
}
```

Run the perl script as follows:

`perl test_pattern_2.pl`

and check on the results you get.

You can also search for a pattern not in a word boundary using the `\B` escape. With this, `/\Bif/` will match only when the characters `i` and `f` occur inside a word and not at the start of a word.

Below is a list of character class codes:

Code	**Equivalent character class**	**What it means**
`\d`	`[0-9]`	Any digit
`\D`	`[^0-9]`	Any character not a digit
`\w`	`[0-9a-zA-Z]`	Any word character
`\W`	`[^0-9a-zA-Z]`	Any character not a word character
`\s`	`[ \t\n\r\f]`	whitespace (space, tab, newline, carriage return, form feed)
`\S`	`[^ \t\n\r\f]`	Any non-whitespace character

0.4.4 Matching a single character

The `.` (dot) matches any single character except the new line character. At least one character should match.

Example

Type the following example in a file called `test_pattern_3.pl`:

```
#!/usr/bin/perl

$line = "help how are you";

print "line = $line\n";

if ($line =~ /hel.p/) {
  print "hel.p was found\n";
}
else {
  print "hel.p was not found\n";
}

if ($line =~ /he.p/){
  print "he.p was found\n";
}
else {
  print "he.p was not found\n";
}
```

Run the perl script as follows:

`perl test_pattern_3.pl`

and check on the results you get.

0.4.5 Matching any single character zero or one time

`?` (question mark) matches any character zero or one time in a string.

Example

Type the following example in a file called `test_pattern_4.pl`:

```
#!/usr/bin/perl

$line = "help how are you";

print "line = $line\n";

if ($line =~ /hel?p/) {
  print "hel?p was found\n";
}
else {
  print "hel?p was not found\n";
}
```

Run the perl script as follows:

`perl test_pattern_4.pl`

and check on the results you get.

0.4.6 Matching a single character including newline

\\`s` is the same as the `.` (dot) seen above, but it includes the newline \\`n`.

Example

Type the following example in a file called `test_pattern_5.pl`

```
#!/usr/bin/perl

print '\s is same as . but includes \n character' . "\n";

$line = "hel\np how are you";

print "line = $line\n";

if ($line =~ /hel\sp/){
  print 'hel\s was found' . "\n";
}
else {
  print 'hel\s was not found' . "\n";
}
```

Run the perl script as follows:

`perl test_pattern_5.pl`

and check on the results you get.

0.4.7 Matching any single character zero or more times

`*` (star) matches any character zero or more times in a string.

Example

Type the following example in a file called `test_pattern_6.pl`

```
#!/usr/bin/perl

$line = "hello how are you";

print "line = $line\n";

if ($line =~ /hel*o/) {
  print "hel*o was found\n";
}
else {
  print "hel*o was not found\n";
}
```

Run the perl script as follows:

`perl test_pattern_6.pl`

and check on the results you get.

0.4.8 Matching any single character one or more times

`+` (plus) matches any character one or more times in a string.

Example

Type the following example in a file called `test_pattern_7.pl`

```
#!/usr/bin/perl

$line = "hello how are you";
```

```
print "line = $line\n";

if ($line =~ /hel+o/) {
  print "hel+o was found\n";
}
else {
  print "hel+o was not found\n";
}
```

Run the perl script as follows:

`perl test_pattern_7.pl`

and check on the results you get.

0.4.9 Matching a grouping of strings

If you are looking for two strings (e.g. `this` and `that`), you can do it as follows:

```
if (($in =~ /this/) || ($in =~ /that/)) {
...
}
```

You can group the matching as follows:

```
if ($in =~ /this|that/) {
...
}
```

or even do better as follows:

```
if ($in =~ /th(is|at)/) {
...
}
```

Below is a complete example that has been picked from the site shown in the page legend[6].

Example

Type the following example in a file called `numspeller2.pl`

```
#!/usr/bin/perl -w
# numberspeller:  prints out word approximations of numbers
# simple version, only does single-digits

$exit = "";  # whether or not to exit the script.

while ($exit ne "n") {

    while () {
        print 'Enter the number you want to spell(0-9): ';
        chomp($_ = <STDIN>);
        if (/^\d$/) {
            print "Thanks!\n";
            last;
        } elsif (/^$/) {
            print "You didn't enter anything.\n";
        } elsif (/\D/) {        # nonnummbers
```

[6]http://work.lauralemay.com/samples/perl.html

```
            if (/[a-zA-z]/) { # letters
                print "You can't fool me.  There are letters in there.\n";
            } elsif (/^-\d/) { # negative numbers
                print "That's a negative number.  Positive only, please!\n";
            } elsif (/\./) { # decimals
                print "That looks like it could be a floating-point number.\n";
                print "I can't spell a floating-point number.  Try again.\n";
            } elsif (/[\W_]/) {  # other chars
                print "huh?  That *really* doesn't look like a number\n";
            }
        } elsif ($_ > 9) {
            print "Too big!  0 through 9, please.\n";
        }
    }

    print "Number $_ is ";
    /1/ && print 'one';
    /2/ && print 'two';
    /3/ && print 'three';
    /4/ && print 'four';
    /5/ && print 'five';
    /6/ && print 'six';
    /7/ && print 'seven';
    /8/ && print 'eight';
    /9/ && print 'nine';
    /0/ && print 'zero';
    print "\n";

    while () {
        print 'Try another number (y/n)?: ';
        chomp ($exit = <STDIN>);
        $exit = lc $exit;
        if ($exit =~ /^[yn]/) {
            last;
        }
        else {
            print "y or n, please\n";
        }
    }
}
```

Run the perl script as follows:

```
perl numberspeller.pl
```

and check on the results you get. Check that whenever you provide a number in the range [0 - 9], the number is converted to a string and printed out on the standard output. Try to provide different combinations and check on the output you get.

0.5 Debugging Perl scripts - The Perl Debugger

Nobody writes perfect code on the first trial. It is thus necessary to have a debugger that helps you go into your code line by line, examine the various variables, registers & memory locations and make sure you're getting the results you should.

In the previous experiments that dealt with the C-language, you encountered the `gdb` debugger that helps you debug C-programs.

In Perl, the debugger is not a separate program as it is in the typical compiled environment like C-language. Instead, the `-d` flag tells the compiler to insert source information into the parse trees it's about to hand off to the interpreter. That means your code must first compile correctly for the debugger to work on it. The debugger won't run until you have fixed all compiler errors.

You'll find below 10 steps to debug Perl programs. These steps have been taken from a site where the link is mentioned in the legend below[7].

To understand the perl debugger commands in detail, let us create the following sample perl program (`perl_debugger.pl`).

Example

Type the following example in a file called `perl_debugger.pl`

```
#!/usr/bin/perl -w

# Script to list out the filenames (in the pwd) that contains specific pattern.

#Enabling slurp mode
$/=undef;

# Function : get_pattern
# Description : to get the pattern to be matched in files.
sub get_pattern
{
  my $pattern;
  print "Enter search string: ";
  chomp ($pattern = <STDIN>);
  return $pattern;
}

# Function : find_files
# Description : to get list of filenames that contains the input pattern.
sub find_files
{
  my $pattern = shift;
  my (@files,@list,$file);

  # using glob, obtaining the filenames,
  @files = <./*>;

  # taking out the filenames that contains pattern.
  @list = grep {
    $file = $_;
    open $FH,"$file";
    @lines = <$FH>;
    $count = grep { /$pattern/ } @lines;
    $file if($count);
  } @files;
  return @list;
}
```

[7]http://www.thegeekstuff.com/2010/05/perl-debugger/

```
# to obtain the pattern from STDIN
$pattern = get_pattern();

# to find-out the list of filenames which has the input pattern.
@list = find_files($pattern);

print join "\n", @list;
```

0.5.1 Enter Perl Debugger

Run the Perl script as follows:

```
perl -d perl_debugger.pl
```

It should promt:

```
DB<1>
```

0.5.2 View specific lines or subroutine statements using (l)

The command `l` takes a line number or a subroutine name. If provided with a line number, it displays the content of that line. If given a subroutine name, it displays the lines of code that correspond to that subroutine.

Execute the command `l 10` as follows:

```
DB<1> l 10
```

It should display:

```
10:  my $pattern;
```

Now execute the command `l get_pattern` as follows:

```
DB<2> l get_pattern
```

it displays the following:

```
11      {
12:       my $pattern;
13:       print "Enter search string: ";
14:       $pattern = <STDIN>;
15:       chomp ($pattern);
16:       return $pattern;
17      }
```

0.5.3 Set the breakpoint on find_files function using (b)

Execute the command:

```
DB<3> b find_files
```

0.5.4 Set the breakpoint on specific line using (b)

Execute the command:

```
DB<4> b 44
```

0.5.5 View the breakpoints using (L)

Execute the command:

```
L
```

You should get the following output:

```
DB<5> L
perl_debugger.pl:
 23:        my $pattern = shift;
   break if (1)
 44:    @list = find_files($pattern);
   break if (1)
```

0.5.6 step by step execution using (s and n)

You can use the command `n` (next) or `s` (step) in a similar way you were used to with the `gdb` debugger.

```
DB<5> s
 main::(./perl_debugger.pl:39): $pattern = get_pattern();

DB<5> s
 main::get_pattern(./perl_debugger.pl:12):
 12: my $pattern;
```

0.5.7 Continue till next breakpoint (or line number, or subroutine) using (c)

You can use the command `c` to continue execution to the next breakpoint as follows:

```
DB<5> c
Enter search string: perl
main::find_files(./perl_debugger.pl:22):
22: my $pattern = shift;
```

0.5.8 Continue down to the specific line number using (c)

```
DB<5> c 36
main::find_files(./perl_debugger.pl:36):
36: return @list;
```

0.5.9 Print the value of a specific variable using (p)

You can use the command `p` to print the value of a variable as follows:

```
DB<6> p $pattern
```

0.5.10 Restart execution using (R) and quit using (q)

You can restart execution of your Perl script from the first instruction by using the command `R` or quit the debugger by using the command `q`.

0.5.11 Get debug commands from the file (source)

Perl debugger can get debug commands from a file and execute them. For example, create the file called `debug_cmds` with the perl debug commands as follows:

```
c
p $pattern
q
```

For example, restart the execution of the Perl script with the debugger as follows:

```
perl -d perl_debugger.pl
```

and then type the following:

```
source debug_cmds
```

Note that the debugger commands will get executed line by line.

0.5.12 Summary of perl debugger commands

- `h` or `h h` for help page
- `c` to continue down from current execution till the breakpoint otherwise till the subroutine name or line number
- `p` to show the values of variables
- `b` to place the breakpoints
- `L` to see the breakpoints set
- `d` to delete the breakpoints
- `s` to step into the next line execution
- `n` to step over the next line execution, so if next line is subroutine call, it would execute subroutine but not descend into it for inspection
- `source ‘‘file’’` to take the debug commands from the file
- `l ‘‘subname’’` to see the execution statements available in a subroutine
- `q` to quit from the debugger mode

0.6 Formats

Perl provides the notion of a simple report writing template, called a *format*. A format defines a constant part (the column headers, labels, fixed text, or whatever) and a variable part (the current data you're reporting)[8].

Using a format consists of doing three things:

- Defining a format.
- Loading up the data to be printed into the variable portions of the format (fields).
- Invoking the format.

We'll explain formats using a concrete example.

Examples

Type the following example in a file called `test_format_1.pl`:

[8] Randal L. Schwartz, Tom Christiansen and Larry Wall - Learning Perl, 2nd Edition

```
#!/usr/bin/perl

format ADDRESSLABEL =
===============================
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<<<<<<<<<<< |
$address
| @<<<<<<<<<<<<<<<<, @< @<<<< |
$city,          $state, $zip
===============================
.

open(ADDRESSLABEL, ">labels-to-print") || die "can't create";
open(ADDRESSES, "addresses.txt") || die "cannot open addresses";

while (<ADDRESSES>) {
    chomp; # remove newline
    ($name, $address, $city, $state, $zip) = split(/:/);
        # load up the global variables
    write (ADDRESSLABEL); # send the output
}
```

Open a text file and type down the following:

```
Mohammad Asmar:44 Ma3ahed Street:Ramallah:Ramallah & Bireh Municipality:P.O.Box: 14
Mervat Shaheen:3 rawda Street:Jenin:Northern Municipality:P.O.Box: 260
Stonehenge:4470 SW Hall Suite 107:Beaverton:OR:97005
Fred Flintstone:3737 Hard Rock Lane:Bedrock:OZ:999bc
```

Save the file as `addresses.txt`.

Run the Perl script as follows:

```
perl test_format_1.pl
```

Open the file called `labels-to-print` and check on its content. That is what we call data formatting.

In the above example, `format` is a keyword and `ADDRESSLABEL` is the name of the format. What follows the format name is called a *template*. The end of the template is indicated by a line consisting of a single dot by itself. Note that templates are sensitive to whitespaces.

The template definition contains a series of *fieldlines*. Each fieldline may contain fixed text which is that will be printed out literally when the format is invoked.

0.6.1 Text Fields

Fieldlines may also contain `fieldholders` for variable text. If a line contains fieldholders, the following line of the template (called the value line) dictates a series of scalar values - one per fieldholder - that provide the values that will be plugged into the fields.

As an example, the fieldholder `@<<<<<<<<<<` specifies a left-justified text field with 11 characters.

If the characters following the `@` are left-angle brackets (`<<<<`), you get a left-justified field; that is, the value will be padded on the right with spaces if the value is shorter than the field width. If the characters following the `@` are right-angle brackets (`>>>>`), you get a right-justified field - that is, if the value is too short, it gets padded on the left with spaces.

Finally, if the characters following the `@` are vertical bars (`||||`), you get a centered field: if the value is too short, it gets padded on both sides with spaces, enough on each side to make the value mostly centered within the field.

0.6.2 Numeric Fields

Another kind of fieldholder is a fixed-precision numeric field, useful for those big financial reports. This field also begins with `@`, and is followed by one or more `#`'s with an optional dot (indicating a decimal point). Once again, the `@` counts as one of the characters of the field. For example:

```
format MONEY =
Assets: @#####.## Liabilities: @#####.## Net: @#####.##
$assets, $liabilities, $assets-$liabilities
.
```

Examples

Type the following example in a file called `test_format_2.pl`:

```
#!/usr/bin/perl

format MONEYLABEL =
Assets: @#####.## Liabilities: @#####.## Net: @#####.##
$assets, $liabilities, $assets-$liabilities
.
open(MONEYLABEL, ">money-formated") || die "can't create";
open(MONEY, "money.txt") || die "cannot open addresses";
while (<MONEY>) {
    chomp; # remove newline
    ($assets, $liabilities) = split(/:/);
        # load up the global variables
    write (MONEYLABEL); # send the output
}
```

Open a text file and type down the following:

```
1.3456:2.01111
222222.345:9898
```

Save the file as `money.txt`.

Run the Perl script as follows:

```
perl test_format_2.pl
```

Open the file called `money-formated` and check on its content.

0.7 Packages Modules and Libraries

A package is a collection of code which lives in its own namespace. As such, Perl uses packages to partition the global namespace. Just as directories contain files, packages contain identifiers. Every global identifier (variables, functions, file and directory handles, and formats) has two parts: its package name and the identifier proper. These two pieces are separated from one another with a double colon. For example, the variable `$CGI::needs_binmode` is a global variable named `$needs_binmode`, which resides in package `CGI`.

Example

Type the following example in a file called `test_package_1.pl`:

```
#!/usr/bin/perl

package Alpha;
my $aa = 10;
   $x = "azure";

package Beta;
my $bb = 20;
   $x = "blue";

package main;
print "$aa, $bb, $x, $Alpha::x, $Beta::x\n";
```

Run the Perl script as follows:

```
perl test_package_1.pl
```

and check on the output you get. Explain the results that you get.

As you can see from the previous example, `package` is a compile-time declaration that sets the default package prefix for unqualified global identifiers. This effect lasts until the end of the current scope (a brace-enclosed block, file, or eval). The effect is also terminated by any subsequent package statement in the same scope. All programs are in package `main` until they use a `package` statement to change this.

0.7.1 Modules

The unit of software reuse in Perl is the *module*, a file that has a collection of related functions designed to be used by other programs and library modules. Every module has a public interface, a set of variables and functions that outsiders are encouraged to use.

The `require` or `use` statements both pull a module into your program, although their semantics are slightly different. The keyword `require` loads modules at runtime, with a check to avoid the redundant loading of a given module. The keyword `use` is like `require`, with two added properties: compile-time loading and automatic importing.

The required file extension for a Perl module is ".pm". The module named `FileHandle` would be stored in the file `FileHandle.pm`. The full path to the file depends on your include path, which is stored in the global `@INC` variable.

Import/Export Regulations

The following is a typical setup for a hypothetical module named `Cards::Poker` that demonstrates how to manage its exports. The code goes in the file named `Poker.pm` within the directory `Cards`: that is, `Cards/Poker.pm`. Here's that file, with line numbers included for reference[9]:

```
1    package Cards::Poker;
2    use Exporter;
3    @ISA = ('Exporter');
4    @EXPORT = qw(&shuffle @card_deck);
5    @card_deck = ();                      # initialize package global
6    sub shuffle { }                       # fill-in definition later
7    1;                                    # don't forget this
```

- Line 1 declares the package that the module will put its global variables and functions in. Typically, a module first switches to a particular package so that it has its own place for global variables and functions, one that won't conflict with that of another program.

[9]Tom Christiansen & Nathan Torkington - Perl Cookbook - 1st Edition

This *must* be written exactly as the corresponding use statement will be written when the module is loaded.

- Line 2 loads in the Exporter module, which manages your module's public interface as described below. Line 3 initializes the special, per-package array `@ISA` to contain the word `''Exporter''`. When a user says use `Cards::Poker`, Perl implicitly calls a special method, `Cards::Poker->import()`. You don't have an `import` method in your package, but that's OK, because the Exporter package does, and you're *inheriting* from it because of the assignment to `@ISA` (is a). Perl looks at the package's `@ISA` for resolution of undefined methods.
- Line 4 assigns the list (`'&shuffle'`, `'@card_deck'`) to the special, per-package array `@EXPORT`. When someone imports this module, variables and functions listed in that array are aliased into the caller's own package. That way they don't have to call the function `Poker::Deck::shuffle(23)` after the import. They can just write `shuffle(23)` instead. This won't happen if they load `Cards::Poker` with `require Cards::Poker`; only a `use` imports.
- Lines 5 and 6 set up the package global variables and functions to be exported. You're free to add other variables and functions to your module as well, including ones you don't put in the public interface via `@EXPORT`.
- Finally, line 7 is a simple 1, indicating the overall return value of the module. If the last evaluated expression in the module doesn't produce a true value, an exception will be raised.

More keywords that you can use while building your modules are:

- `$VERSION`: When a module is loaded, a minimal required version number can be supplied. If the version isn't at least this high, the `use` will raise an exception.

  Example

  ```
  use YourModule 1.86; # If $VERSION < 1.86, fail
  ```

- `@EXPORT`: The array contains a list of functions and variables that will be exported into the caller's own namespace so they can be accessed without being fully qualified.

  Example

  ```
  @EXPORT = qw( F1  F2 @List);
  ```

  Thus the functions `F1`, `F2` and the array `@List` are exported.

- `@EXPORT_OK`: This array contains symbols that can be imported if they're specifically asked.

  Example

  ```
  @EXPORT_OK = qw(Op_Func %Table);
  ```

  Then the user could load the module as follows:

  ```
  use YourModule qw(Op_Func %Table F1);
  ```

  To get everything in `@EXPORT` plus extras from `@EXPORT_OK`, use the special `:DEFAULT` tag, such as:

  ```
  use YourModule qw(:DEFAULT %Table);
  ```

- %EXPORT_TAGS: This hash is used by large modules like CGI or POSIX to create higher-level groupings of related import symbols. Its values are references to arrays of symbol names, all of which must be in either @EXPORT or @EXPORT_OK.

  Example

  ```
  %EXPORT_TAGS = (
          Functions => [ qw(F1 F2 Op_Func) ],
          Variables => [ qw(@List %Table)  ],
  );
  ```

  An import symbol with a leading colon means to import a whole group of symbols. Here's an example:

  ```
  use YourModule qw(:Functions %Table);
  ```

Below is a complete example on how to write and call your own module.

Example

Type the following in a file called Foo.pm:

```
#!/usr/bin/perl

package Foo;
sub bar {
   print "Hello $_[0]\n"
}

sub blat {
   print "World $_[0]\n"
}
1;   # Don't forget the 1 which indicates the return value is TRUE
```

Type the following example in a file called test_foo_require.pl:

```
#!/usr/bin/perl

require Foo;

Foo::bar("a");
Foo::blat("b");
```

Run the perl script as follows:

```
perl test_foo_require.pl
```

and check on the results you get.

Type the following example in a file called test_foo_use.pl:

```
#!/usr/bin/perl

use Foo;

bar("a");
blat("b");
```

Run the perl script as follows:

```
perl test_foo_use.pl
```

and check on the results you get. Explain in particular why it doesn't work.

<u>**Example**</u>

Type the following in a file called `Foo_complete.pm`:

```
package Foo_complete;

require Exporter;
@ISA = qw(Exporter);
@EXPORT = qw(bar blat);

sub bar { print "Hello $_[0]\n" }
sub blat { print "World $_[0]\n" }
sub splat { print "Not $_[0]\n" }  # Not exported!

1;
```

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

```
#!/usr/bin/perl

use Foo_complete;

bar("a");
blat("b");
splat("c");
```

Run the perl script as follows:

```
perl test_foo_complete_use.pl
```

and check on the results you get. Explain why it works now for the first 2 subroutines but not for the third subroutine.

Modules freely available

CPAN, the Comprehensive Perl Archive Network, is a gigantic repository of nearly everything about Perl you could imagine, including source, documentation, alternate ports, and above all, modules. Before you write a new module, check with CPAN to see whether one already exists that does what you need. Even if one doesn't, something close enough might give you ideas.

You can access CPAN at `http://www.cpan.org`. The module directory itself can be visited at `http://www.cpan.org/modules`. It contains indices of all registered modules plus three convenient subdirectories: *by-module*, *by-author*, and *by-category*. All modules are available through each of these, but the *by-category* directory is probably the most useful. There you will find directories covering specific applications areas including operating system interfaces; networking, modems, and interprocess communication; database interfaces; user interfaces; interfaces to other programming languages; authentication, security, and encryption.

0.7.2 Libraries

A library is a collection of loosely related functions designed to be used by other programs. It lacks the rigorous semantics of a Perl module. The extension is usually .pl like any other Perl script.

Perl libraries can be loaded by inserting the following line `require lib.pl` (assuming the library name is `lib.pl`).

Libraries work well when used by a program, but problems can arise when libraries use one another. Consequently, simple Perl libraries have been rendered mostly obsolete, replaced by the more modern modules. But some programs still use libraries.

0.7.3 The include path for Perl modules

Perl interpreter is compiled with a specific `@INC` default value. To find out this value of `@INC`, run the following command:

```
perl -e 'for (@INC) {printf "%d %s\n", $i++, $_}'
```

If the module you intend to use is located under one of the directories mentioned in the `@INC` variable, you can use the module in your Perl scripts by including the below statement:

```
use myModule
```

If the module is installed elsewhere, you can set the environment variable `PERL5LIB` (if it is not defined, `PERLLIB` is used) on your shell. Perl pre-pends `@INC` with a list of directories (colon-separated) contained in `PERL5LIB`.

A third way to include modules installed in non-standard locations, you can use the `-I` option when executing the following command on the shell as follows:

```
perl -I /my/moduledir your_script.pl
```

A fourth way would be to include the following line as the first line in a perl script:

```
#!/usr/local/bin/perl -w -I /my/moduledir
```

0.7.4 Preparing a Module for Distribution

If you want to prepare your module to get installed in a standard distribution format so you can easily send your module to a friend, you can use the command `h2xs` as follows:

```
h2xs -XA -n myModule
```

The above will create a folder called `myModule` that will contain the following:

- Folder `lib`
- Folder `t`
- A file called `Changes`
- A file called `Makefile.PL`
- A file called `MANIFEST`
- A file called `README`

You can afterwards tar and gzip the folder `myModule` as follows:

```
tar cf myModule.tar myModule
gzip myModule.tar
```

The above commands will generate the file `myModule.tar.gz` that you can distribute.

Once your friends and colleagues receive your tarred and gzipped file, they can install your module `myModule` by executing the below commands in sequence:

```
gzip -dc myModule.tar.gz | tar -xvf -
cd myModule
perl Makefile.PL
make
make install
```

Note that the above installation procedure will install the module myModule under the standard location:

```
/usr/local/lib/perl5/site_perl/5.8/i386-...
```

The above signifies that you have root privilege.

If you don't have root privilege, you can do the module installation under your home directory for example by using the PREFIX compilation directive as follows:

```
perl Makefile.PL PREFIX=/home/user/folder LIB=/home/user/folder
```

Example

We'll use for the current example the file Foo.pm that was created previously. Execute the following commands in sequence:

```
cd ~
mkdir tmp
cp Foo.pm tmp/.
cd tmp
h2xs -XA -n Foo
```

Note that the folder Foo gets created under directory tmp. At this stage, you can tar and gzip the folder Foo and send it to your colleagues as mentioned previously.

If you wish to install the module Foo, you can execute the following commands is sequence:

```
perl Makefile.PL
make
make install
```

If you do not have the root privilege, you can install the module Foo under folder tmp as follows:

```
perl Makefile.PL PREFIX=/home/user/tmp LIB=/home/user/tmp
```