Perl Scripting

Agenda

- **Overview of Perl**
- **Datatypes in Perl**
- **Arrays and Hashes**
- **Branching and Looping**
- **Subroutines**

Objectives

At the end of this module you will be able to:

- Know different datatypes supported by Perl and how to use them
- Perform operations on variables
- Use arrays and hashes
- Use various branching and looping constructs
- Define subroutines that take arguments and return a value

Overview of Perl

Objectives

At the end of this module you will know:

- What is Perl
- Why to use Perl
- How to get Perl
- Where to use Perl
- Writing a Perl script and executing it

About Perl

- Perl stands for **P**ractical **E**xtraction and **R**eporting **L**anguage
- It was developed by Larry Wall, Ver 1.0 released in 1987
- As the name implies, it is used for data extraction, manipulation and reporting.
- Apart from that, Perl being a general purpose programming language, can be used for any kind of programming tasks such as
  - Reporting
  - Data validation
  - Network/web/report automation

- Perl can serve as a glue language.

Why to use Perl & Where to use Perl

- Perl is one of the widely used scripting languages for following reasons
  - Easy to learn and program
  - Can be used to solve simple to complex tasks
  - Available across operating system platforms (e.g., Windows, UNIX/Linux flavours)
  - Portable
  - Perl is freeware
  - Extensive support for regular expressions
  - Large collection of Perl Modules (available at www.cpan.org)

- Prominent uses of Perl is for …
  - System administration
  - Test automation
  - Bio-informatics
  - Web development

Where to get Perl

- Perl versions
  - Perl 6.x is the latest and widely used.
  - However, Perl 5.x is still in good use.

- To know which version of Perl is installed, run the command

  perl -v

- Perl binaries can be downloaded from the following
  - ActiveState Perl (www.activestate.com/activeperl/downloads)
    - Has versions for both UNIX and Windows
  - Strawberry Perl (http://strawberryperl.com/)
    - provides Perl for Windows platform

Developing application in Perl

- A plain text editor is good enough to create Perl script.
- GUI-based front-ends do exist for various OS platforms to create Perl scripts.
- A Perl script file is give .pl or .plx extension, not necessary, but for easy identification.

The First Perl Program

```
1. #!/usr/local/bin/perl

2. # Filename : greet.pl

3. print "Enter Your Name : " ;     # show the prompt

4. $Name = <stdin>;                 # Reading input from keyboard

5. # print the greeting

6. print "Hello  $Name  - Welcome to the world of PERL !!!" ;
```

Note: Line numbers are not part of the script, used for reference in subsequent slider .

How to run the perl script in UNIX/Linux

- In UNIX/Linux run the perl command with the filename as argument

```
$ perl greet.pl
Enter Your Name : victor
Hello  victor
  - Welcome to the world of PERL !!!
$
```

- Alternately, set the execute attribute of the script file, and then run the program as shown here.

```
$ chmod  +x  greet.pl
$ ./greet.pl
Enter Your Name : victor
Hello  victor
  - Welcome to the world of PERL !!!
$
```

How to run the perl script in Windows

- In windows environment as well it is the same may – run perl command with the filename as argument

```
C:\perlex> perl  greet.pl

Enter Your Name : victor

Hello  victor

  - Welcome to the world of PERL !!!

C:\perlex>
```

Dissecting the First Perl Program

1. **#!/usr/local/bin/perl**

   The line that begins with the special pattern “#!” line indicates that the script that follows this has to be executed the command specified next to “#!”.

   This being a perl script, path for perl is specified here

2. **# Filename : greet.pl**

   Text that follows # is considered as a comment.

3. **print "Enter Your Name : " ; # show the prompt**

   print command displays text onto the screen

Note that every Perl statement must end with a semicolon like in C

Note the way comment is added.

A comment can appear as part of a perl statement.

Dissecting the First Perl Program (Contd.).

4. **$Name = <stdin>;** **# Reading input from keyboard**

   this statement reads input from the standard input (default keyboard) and stored into the variable “Name”. Note that the variable name should be prefixed with $.

   For now follow this, more on this in upcoming slides

5. **# print the greeting**

6. **print "Hello $Name - Welcome to the world of PERL !!!" ;**

   Note again the variable is prefixed with $ symbol

Data types in Perl

Agenda

- At the end of this module you will know:
  - Different datatypes in Perl
  - Using scalars and scalars with string values
  - How to use Quote operators
  - Interpolation
  - The Numeric and string context
  - How to use CMP and range operators

Datatypes in Perl

- Perl supports various data types
- Perl does not mandate a variable to be defined before it is used.
- Perl recognizes the type of a variable by the prefix it has
- Different prefixes indicate different types of variable
- Broad classification is
  - Scalar → Variable that holds only single value
  - Array → Variable that holds multiple value of same kind
  - Associative array → Variable that holds an array of (key, value) pairs
  - Glob → Variable that holds address of another variable
  - File handle → holds handle of an open file

- Note that there is no special syntax to indicate if a variable holds integer data or float or string

Datatypes in Perl (Contd.)

Prefix	What it implies	Example
$	Scalar	$count = 0; $nums="1,2,3,4";
@	Array	@nums=(1, 2, 3, 4);
%	Associative Array	%nums=(one , 1, two, 2, three, 3);

Variables

- In Perl names of variables are case sensitive

  Thus, TotalCost and totalcost are names of two different variables

- Names of variables can not have any special characters other than underscore (_).

- Variables need not be predeclared

- No limit on the length of a variable but ideal to restrict it to 30 characters

Scalars

- Scalars are variables that can hold a single value.
- A variable that is prefixed with $ symbol is considered as scalar.

Examples:

```
$count = 0;
$pi = 3.142;
$msg = “Operation success”;
```

Numerical Scalars

- The adjacent example shows the usage of numeric scalars.
- Note that the first print produces the output 0 1 3
- Print command does not add a newline character by default to the output.
- So, the output of the next print command, which is 4 (result of the expression) appears on the same line of the previous output.

```
# filename : numscalar.pl
$x=0;
$y = $x + 1;
$z = $y + $x + 2;
print "$x $y $z";
print $x + $y + $z;
```

```
$ perl numscalar.pl
0 1 34
$
```

Scalars with fractional values

- The adjacent example shows the usage of scalars with floating values.
- Note that the spaces around operators is not mandatory.
- In the print statement, \n is included to force newline character.

```
# filename : numfract.pl
$a=1.1;
$b=2.0002;
$c=$a+$b;
$d=$a / $b;
print "$c   $d\n";
```

```
$ perl   numfract.pl
3.1002   0.544554455445545
$
```

String Scalars

- A character sequence surrounded by single quotes or double quotes is considered as a string literal.

"hello" 'format' etc.

- A variable that holds a character sequence is called string scalar.

Single quote and double quote

- For a single quoted string interpolation does not take place.
- For a double quoted string interpolation takes place.

```
# filename : string.pl
$name = "victor";
print ' **** in single quote Hello $name. Good evening. **** \n' ;
print " === in double quote Hello $name. Good evening===. \n" ;
```

```
$ perl   string.pl
 **** in single quote Hello $name. Good evening. **** \n
=== in double quote Hello victor. Good evening ===.
$
```

Quote operators

- It is not necessary to use single quote or double quote to define strings.
- Programmer can choose any other character for the purpose.
- Perl provides two types of quote operators
  - q → for single quoting a string, but using any other character
  - qq → for double quoting a string, but using any other character

Examples:

```
$x = q   : hello world :                    # using : instead of ‘
print $x
```

Quote operators (Contd.).

- Why the output is not as expected?

```
# file : quotes1.pl
$s="script";
$nl="\n";
$x=q(perl-$s);
$y=qq(perl-$s);
print "$x $nl";
print "$y $nl";
```

```
$ perl   quotes1.pl
perl-$s
perl-script
$
```

Quote operators (Contd.).

```
# file  quote.pl
$str0 = hello;
$str1 = q(hello);
$str2 = qq(hello);

$str3 = q(hello $$);
$str4 = qq(hello $$);
$str5 = qq(hello\$);
$str6 = qq(hello$$);
print "str0 = $str0\n";
print "str1 = $str1\n";
print "str2 = $str2\n";

print "str3 = $str3\n";
print "str4 = $str4\n";
print "str5 = $str5\n";
print "str6 = $str6\n";
```

```
$ perl quote.pl
str0 = hello
str1 = hello
str2 = hello
str3 = hello $$
str4 = hello 184
str5 = hello$
str6 = hello184
$
```

Note: $$ represents process id.

Quote operators (Contd.).

```
# file : quote3.pl

$x=qq("some text in double quotes", 'text in single quotes' );

$y="\"some text in double quotes\", \'text in single quotes\'";

print "$x\n";

print "$y\n";
```

```
$ perl quote3.pl

"some text in double quotes", 'text in single quotes'

"some text in double quotes", 'text in single quotes‘

$
```

Back quote and qx operator

- To execute a command inside perl script the command along with arguments if any has to be enclosed in back quotes (` `)
- Alternately use **qx** operator

```
# filename : backquote1.pl
$file="backquote1.pl";
print `type $file`;
```

```
# filename : backquote2.pl
$file="backquote2.pl";
print qx(type $file);
```

```
$ perl backquote1.pl
# filename : backquote1.pl
$file="backquote1.pl";
print  `type  $file`;
$
```

```
$ perl backquote2.pl
# filename : backquote2.pl
$file="backquote2.pl";
print qx(type  $file);
$
```

defined function & undef value

- An uninitialized scalar variable is assigned with a special value *undef*
- It is also possible to undef a scalar which has been defined earlier.
- ***defined*** function is used to test if a scalar is defined or not.

```
if (defined $x)  {  print "x is defined\n"; }
else {  print "x is undefined\n";  }

$x=20;

if (defined $x)  {  print "x is defined\n"; }
else {  print "x is undefined\n";  }

$x=undef;

if (defined $x)  {  print "x is defined\n"; }
else {  print "x is undefined\n";  }
```

Output

```
x is undefined
x is defined
x is undefined
```

Interpolation

- Interpolation of a scalar variable depends on whether the variable appears in single quotes or double quotes.
- interpolation takes place for scalar variables when enclosed in double quotes wherein variable is substituted by its value.
- Interpolation of variables will not take place for text enclosed in singe quotes.
- Similarly escape sequence such as "\n" is interpolated only if it is in double quotes.

```
$x=200;
print 'value of x = $x\n';
print "value of x = $x\n";
```

Output

```
value of x = $x\nvalue of x = 200
```

String operations

- Concatenation of strings is possible using dot (.) operator.

```
$x = "abc" . "def";
$y = "hello";
$y .= "world";          # equivalent to   $y = $y  . "world"
print  "$x\n";
print  "$y\n";
```

Output

```
abcdef
helloworld
```

String operations (Contd.).

- length() → returns the length of a string
- substr() → returns the substring of a string
  - The function takes the source string, start index and number of characters to extract which is optional
  - Index can be –ve, in which case it indicates position from the end of the string. Index value for the last character is -1

```
$x='';
$y="";
$p="perl program";
print length($x) . "\n";
print length($y) . "\n";
print length($p) . "\n";
print substr($p, 0, 4)  . "\n";  # extract first 4 characters
print substr($p, 5)  . "\n";  # extract from 6th char to end of the string
print substr($p, -3)  . "\n"; # extract last three chars of the string
```

Output

```
0
0
12
perl
program
ram
```

String operations (Contd.).

- uc() / lc()
  - returns string in uppercase / lowercase
- ucfirst() / lcfirst()
  - returns string in which only the first character is in uppercase / lowercase

```
$x="perl Program";
$y="LANGUAGE";
$xu = uc($x);
$xl = lc($y);
$xuf = ucfirst($x);
$ylf = lcfirst($y);
print "$x\n";
print "$y\n";
print "$xu\n";
print "$xl\n";
print "$xuf\n";
print "$ylf\n";
```

Output

```
perl Program
LANGUAGE
PERL PROGRAM
language
Perl Program
lANGUAGE
```

Numeric Arithmetic Operations

- Perl supports all numeric operators as in C
- In addition, Perl has exponent operator (**)

```
$x=5;
$y=4;
$z = ++$x + $y--;
print "\$x = $x   \$y = $y   \$z = $z\n";

++$a;       # perl assumes a is initialized to 0, increments a to 1
--$b;       # perl assumes b is initialized to 0, decrements b to -1
print "\$a = $a   \$b = $b\n";

$z = $x ** $y;          # performs x to the power of y
print "\$x = $x   \$y = $y   \$z = $z\n";
```

Output

```
$x = 6   $y = 3   $z = 10
$a = 1   $b = -1
$x = 6   $y = 3   $z = 216
```

Numeric context

- Numeric context is applied when arithmetic operators are used.
- In numeric context
  - Strings are converted to numeric values
  - Undefs are evaluated to zero
  - Array is evaluated to its length

```
$a=5.13;
$b=10;
$c = $a + $b;
print "$c\n";

$m="abc";
$n="pqr";
$d = $m + $n;
print "$d\n";
```

Output

```
15.13

0
```

```
$p="5ab";
$q="30def";
$f = $p + $q;
print "$f\n";

$x=undef;
$t = $x + $y;
print "$t\n";
```

Output

```
35
0
```

String context

- String context is applied when string operations such as dot operation (.) or double quotes (“”) are used.
- In string context
  - Numeric values are converted to string equivalent
  - Undefs are evaluated to empty string

```
$m="abc";
$n="pqr";
$p=undef;

$d = $m . $n;
print "$d\n";

$f = $m . $p . $n;
print "$f\n";

$a=5.13;
$b=10;
$c = $a . $b;
print "$c\n";
```

Output

```
abcpqr

abcpqr

5.1310
```

Equality & Relational Operations for Numbers

- Perl supports relational operators as in C for numbers.
- In addition, Perl has operator <=> which return -1, 0 or 1 depending on a < b or a == b or a > b.

```
$x=5;
$y=3;

print "\$x = $x   \$y = $y   $x == $y  ==> ", $x == $y ? TRUE : FALSE, "\n";
print "\$x = $x   \$y = $y   $y == $x  ==> ", $y == $x ? TRUE : FALSE, "\n";
print "\$x = $x   \$y = $y   $x <=> $y  ==> ", $x <=> $y, "\n";
print "\$x = $x   \$y = $y   $y <=> $x  ==> ", $y <=> $x, "\n";
print "\$x = $x   \$y = $y   $x <=> $x  ==> ", $x <=> $x, "\n";
```

Output

```
$x = 5   $y = 3   5 == 3  ==> FALSE
$x = 5   $y = 3   3 == 5  ==> FALSE
$x = 5   $y = 3   5 <=> 3  =S=> 1
$x = 5   $y = 3   3 <=> 5  ==> -1
$x = 5   $y = 3   5 <=> 5  ==> 0
```

Comparison operator for numbers

```
#file : numcmpop.pl
print "Enter 1st num : ";
$x1=<STDIN>;
chomp $x1;
print "Enter 2nd num : ";
$x2=<STDIN>;
chomp $x2;
my $res = ($x1 <=> $x2);
print "res = $res\n";
if ($res == 0) { print "both are identical\n"; }
elsif ($res == 1) { print "$x1 > $x2\n"; }
elsif ($res == -1) { print "$x1 < $x2\n"; }
```

Output

```
$ perl   numcmpop.pl
Enter 1st num : 5
Enter 2nd num : 5
both are identical

$ perl   numcmpop.pl
Enter 1st num : 12
Enter 2nd num : 5
12 > 5

$ perl   numcmpop.pl
Enter 1st num : 5
Enter 2nd num : 12
5 < 12
$
```

Relational Operations for Strings

- Relational operators
  - **lt** (for less than)
  - **gt** (for less than)
  - **le** (for <=)
  - **ge** (for >=)

```
$x1="abf";
$x2="abk";
$x3="abc";
$res = $x1;
if ($x2 gt $res) { $res = $x2; }
if ($x3 gt $res) { $res = $x3; }
print $res . "\n";
```

Output

```
abk
```

Equality Operations for Strings

- Equality operators
  - **eq** (for equal), **ne** (for not equal), **cmp** (for comparison)

```
$x1="aba";
$x2=$x1;

if ($x1 eq $x2)  {
   print "\$x1 and \$x2 are identical\n";
}
```

Output

```
$x1 and $x2 are identical
```

```
$x1="abc";
$x2="abcd";

$res=($x1 cmp $x2);
if ($res == 0) { print "$x1  & $x2 are equal\n"; }
elsif ($res == 1) { print "$x1 > $x2\n"; }
elsif ($res == -1) { print "$x1 < $x2\n"; }
```

Output

```
abc < abcd
```

Range operator

- Range operator is used to produce a range of continuous sequence of numbers or characters.
- The range operator result in an array.
- Range operator is in the form

  (*lowerlimit* .. *upperlimit*)

```
@nums=(50..60);
print "@nums" . "\n";

@nums=(50..55,56..60);
print "@nums" . "\n";

@chars=('a' .. 'f');
print "@chars" . "\n";

@chars=('aa' .. 'af');
print "@chars" . "\n";
```

Output

```
50 51 52 53 54 55 56 57 58 59 60

50 51 52 53 54 55 56 57 58 59 60

a b c d e f


aa ab ac ad ae af
```

Arrays and Hashes

Agenda

- At the end of this module you will know:
  - How to declare an array
  - How to slice an array
  - Performing Sort/Reverse/Push/POP operations
  - Interpolation of arrays
  - What is a Hash
  - How to define a Hash
  - Using various functions associated with hash such as
    - Keys and values function
    - Each/Exists/Delete function
  - How to convert from hash to an array
  - What is globes type
  - How to use @ARGV array

Arrays & Lists

- While scalars hold only one value, Perl supports *list* which is a collection of scalars.

- Array is a variable that contains a list.

```
@even = ( 2, 4, 6, 8, 10 );
```

In the above example, *even* is the array name, and (2, 4, 6, 8, 10) the list.

- Array or list can hold numbers or strings or undef values or a mix of different scalars.
- Elements of arrays or lists are accessed specifying index or subscript.
- In Perl, index for array or list starts at zero.

Creating an array

- Here is one way of defining an array, access its elements and print elements of the array
  - The array ***city*** is created as a collection of scalars

```
$city[0] = "Delhi";
$city[1] = "Mumbai";
$city[2] = "Chennai";
$city[3] = "Kolkata";
print ('$city[1] = ', "$city[1]\n");
print  @city;  # o/p without spaces
print "\n";
print  "@city\n";  # o/p with spaces
```

output

```
$city[1] = Mumbai
DelhiMumbaiChennaiKolkata
Delhi Mumbai Chennai Kolkata
```

Note the way to output specific element of the array or entire array elements and use of **@** prefix for city

Creating an array using @

- The following example shows yet another way to define an array
- Points to note:
  - Using **@** prefix for a variable name, indicates the variable should be treated as an array.
  - Individual elements of an array being scalar, they are accessible using the **$** prefix

```
@city = ( "Delhi", "Mumbai", "Chennai", "Kolkata" );

print ('$city[1] = ', "$city[1]\n");   # access an element as a scalar

print ('$city[3] = ', "$city[3]\n");
```

output

```
$city[1] = Mumbai

$city[3] = Kolkata
```

Creating an array using @ (Contd.).

- Elements of a array can be operated individually or collectively.
  - Example below shows how to print all elements of the ***city*** array

```
@city = ( "Delhi", "Mumbai", "Chennai", "Kolkata" );
print ('$city[1] = ', "$city[1]\n");   # access an element as a scalar
print @city;          #  prints all elements without any separator
print "\n";
print "@city\n";   #  prints all elements with space as separator
```

output

```
$city[1] = Mumbai
DelhiMumbaiChennaiKolkata
Delhi  Mumbai  Chennai  Kolkata
```

Note the difference between the two outputs.

Creating an array using @ (Contd.).

- Note that elements in the initializer list need not be in quotes
  - If the string has embedded spaces, it must be in quotes

```
@city = ("New Delhi", Mumbai, Chennai, Kolkata );
print ('$city[0] = ', "$city[0]\n");   # access an element as a scalar
print @city;          #  prints all elements without any separator
print "\n";
print "@city\n";   #  prints all elements with space as separator
```

output

```
$city[0] = New Delhi
New DelhiMumbaiChennaiKolkata
New Delhi  Mumbai  Chennai  Kolkata
```

Note the difference between the two outputs.

Accessing Array Elements

- Array subscript need not be a constant.
  - It can be an expression.

```
@city = ( 'New Delhi', Mumbai,
                Chennai, Kolkata );
$idx=2;
print ('$city[$idx-1] = ', "$city[$idx-1]\n");
print ('$city[$idx] = ', "$city[$idx]\n");
print ('$city[$idx+1] = ', "$city[$idx+1]\n");
```

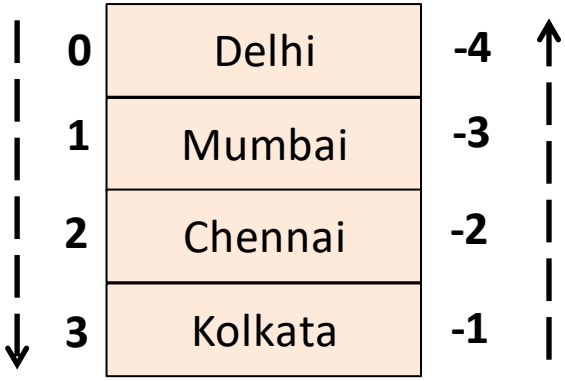
output

```
$city[$idx-1] = Mumbai
$city[$idx] = Chennai
$city[$idx+1] = Kolkata
```

Traversing array in reverse order

- Arrays can be traversed in reverse order.
  - To do that we have to use subscripts with negative values.

```
@city = ('Delhi', 'Mumbai', 'Chennai', 'Kolkata' );
print ('$city[-1] = ', "$city[-1]\n");
print ('$city[-2] = ', "$city[-2]\n");
print ('$city[-4] = ', "$city[-4]\n");
print ('$city[-5] = ', "$city[-5]\n");
```



output

```
$city[-1] = Kolkata
$city[-2] = Chennai
$city[-4] = Delhi
$city[-5] =
```

Note that the subscript -5 is out of bounds, so no output.

Handling subscripts out of bounds will be covered later.

Slicing an Array

- Part of an array (a slice) can be extracted as another array.

```
@days = qw(mon tue wed thu fri sat sun);

@weekdays = @days[0..4];
@weekenddays = @days[5..6];

print "@weekdays\n";
print "@weekenddays\n";
```

output

```
mon tue wed thu fri

sat sun
```

Push and Pop functions

- Push and pop functions are used to insert and remove elements to an array.
- Thus array can be operated as an array.
- Push adds an element to the end of an array
- Pop removes the last element from an array

```
@alpha;
push (@alpha, 'a');
push (@alpha, 'b');
push (@alpha, 'c');
print "@alpha\n";
$x=pop (@alpha);
push (@alpha, 'd');
print "@alpha\n";
```

Output

```
a b c
a b d
```

Sorting an array of strings

- Use **sort** function to sort an array

  sort @arrayname;

- The original array remains unchanged.
- The output of sort has to be saved into an array.
- Sort function, by default sorts in alphabetical order, even if the data is a set of numbers.

```
@cities1 = (Delhi, Kochi, Mumbai, Kolkata, Bangalore, Chennai);
@cities2 = sort @cities1;

$,=' ';     # set ' ’ as output field separator
print @cities1, "\n";
print @cities2, "\n";
```

Output

```
Delhi Kochi Mumbai Kolkata Bangalore Chennai
Bangalore Chennai Delhi Kochi Kolkata Mumbai
```

Sorting string array in reverse order

- To sort a string array in reverse order, apply the following:

  {$b gt $a}

```
@cities1 = (Delhi, Kochi, Mumbai, Kolkata, Bangalore, Chennai);
$,=' ';
print @cities1, "\n";
@cities2 = sort @cities1;
print @cities2, "\n";
@cities2 = (sort {$b gt $a} @cities1);
print @cities2, "\n";
```

Output

```
Delhi Kochi Mumbai Kolkata Bangalore Chennai
Bangalore Chennai Delhi Kochi Kolkata Mumbai
Mumbai Kolkata Kochi Delhi Chennai Bangalore
```

Sorting an array of integers

- Sort function, by default sorts in alphabetical order, even if the data is a set of numbers.
  - Use the expression **{$a <=> $b}** to sort numeric data in ascending order or **{$b <=> $a}** to sort numeric data in descending order.

```
#filename : sort-NumAry.pl
@numary = (9, 7, 12, 5, 8);
$,=' ';
print "original order                    ", @numary,"\n";
print "sorted in dictionary order        ", sort (@numary),"\n";
print "sorted in numerical order         ", (sort {$a <=> $b} @numary), "\n";
print "sorted in reverse numerical order ", (sort {$b <=> $a} @numary), "\n";
```

```
$ perl   sort-NumAry.pl
original order                    9 7 12 5
8
sorted in dictionary order        12 5 7 8
9
sorted in numerical order         5 7 8 9 12
sorted in reverse numerical order 12 9 8 7 5
```

Array Interpolation

- An array variable in double quoted string is interpolated as concatenation of double-quoted strings, by default separated by space or programmer defined delimiter.
- The delimiter can be changed, if required using

$, = "delimiter";

```
@mths_30_days = qw(Apr Jun Sep Nov);
print @mths_30_days, "\n";
print "@mths_30_days\n";
$,="|";   # set |  as delimiter
print @mths_30_days, "\n";
print "@mths_30_days\n";
$,="-"; # set -  as delimiter
print @mths_30_days, "\n";
print "@mths_30_days\n";
```

Output

```
AprJunSepNov
Apr Jun Sep Nov
Apr|Jun|Sep|Nov|
Apr Jun Sep Nov
Apr-Jun-Sep-Nov-
Apr Jun Sep Nov
```

Array Interpolation (Contd.).

- Array interpolation can be avoided using single quotes or prefixing @ with backslash (\) character.

```
@myworld = qw(1 2 3);
print @myworld, "\n";
print "@myworld\n";   # interpolate

# no interpolation of array
print 'contact@myworld.com', "\n";

# interpolate array
print "contact@myworld.com","\n";
# no interpolation of array
print "contact\@myworld.com","\n";
```

Output

```
123
1 2 3
contact@myworld.com
contact1 2 3.com
contact@myworld.com
```

Hashes

- Hash is a compound data type.
- It is also known as associative array.
- A hash holds multiple data elements like an array.
- Hash is an array of ***key-value*** pairs.
- Hash is ordered by key.

- Unlike array operations that use an index which is a number, operations on hashes are performed using a ***key***, which is a string.
  - Hash is addressable by content than by index.

- In a hash, key has to be unique.
- A hash can be empty to begin with and can grow to any size, subject to availability of memory.

- examples of hash are
  - Roll number (key) and score
  - City (key) and capital

RollNo	Score
15020	70
15220	85
15136	90
15142	82
15115	76
15235	70
15200	95
15174	81

Creating a Hash

```
%roll_score = ( 15020 => 70, 15220 => 85,
                                        15136 => 90, 15142 =>
82,
                                        15200 => 95, 15174 =>
81 );

print %roll_score;          # print the hash
print "\n";
$,=' ';                 # set output field separator to space
print %roll_score;          # print the hash
print "\n";
```

RollNo	Score
15020	70
15220	85
15136	90
15142	82
15200	95
15174	81

output

```
15220851502070152009515142821517481151369 0

15220 85 15020 70 15200 95 15142 82 15174 81 15136 90
```

Creating a Hash (contd.).

Alternately, we can create a hash by just placing a sequence of key and value pairs in parenthesis.

```
%roll_score = ( 15020, 70,  15220, 85,  15136, 90,
                15142, 82,  15115, 76,  15235,
70,
                15200, 95,  15174, 81 );
```

RollNo	Score
15020	70
15220	85
15136	90
15142	82
15115	76
15235	70
15200	95
15174	81

Finding size of a hash

To find the size of a hash, follow the following procedure.

1. Get the array of either keys (using **keys** function) or values (using the **values** function)

2. Get the size of the array

```
%roll_scores = ( 15020, 70,
   15220,  85,  15136,  90,
   15200, 95,  15174, 81 );
@rolls = keys %roll_scores;
$size = @rolls;
print "size of roll_scores is $size\n";
```

output

```
size of roll_scores is  5
```

Populating an empty hash

```
$,=' ';

print "***",%roll_score,"^^^\n";

$roll_score{15020} = 70;

print "***",%roll_score,"^^^\n";

$roll_score{15220} = 85;

print "***",%roll_score,"^^^\n";

$roll_score{15136} = 90;

print "***",%roll_score,"^^^\n";
```

output

```
*** ^^^

*** 15020 70 ^^^

*** 15220 85 15020 70 ^^^

*** 15220 85 15020 70 15136 90 ^^^
```

Modifying value of a key in a hash

```
%roll_score = ( 15020, 70,  15220, 85,  15136, 90,
                                        15200, 95,  15174,
81 );
$,=' ';
print %roll_score,"\n";   # print hash before change
$roll_score{15220} = 100;  # change value to 100
$roll_score{15174} = 70;    # change value to 70
print %roll_score,"\n";   # print hash after change
```

Note:

If key exists in the hash, then its value is replaced; otherwise the key with value is added.

output

```
15220 85 15020 70 15200 95 15174 81 15136 90

15220 100 15020 70 15200 95 15174 70 15136 90
```

Removing a key from a hash

```
%roll_score = ( 15020, 70,  15220, 85,  15136, 90, 15200, 95,  15174, 81 );

$,=' ';
print %roll_score,"\n";                    # print the original hash
delete $roll_score{15220};   # remove key 15220
print %roll_score,"\n";                    # print hash after deletion
delete $roll_score{15174};# remove key 15174
print %roll_score,"\n";                    # print hash after deletion
```

output

```
15220 85 15020 70 15200 95 15174 81 15136 90

15020 70 15200 95 15174 81 15136 90

15020 70 15200 95 15136 90
```

Extracting keys of a hash

```
%roll_score = ( 15020, 70,  15220, 85,  15136, 90,
                                        15200, 95,  15174, 81 );

@rollnums = keys %roll_score;
$,=' ';
print  @rollnums,"\n";
```

extract the keys of a hash using **keys** function.
The function returns an array of keys.

output

```
15220 15020 15200 15174 15136
```

Extracting values in a hash

```
%roll_score = ( 15020, 70,  15220, 85,  15136, 90,
                                        15200, 95,  15174, 81 );

@scores = values %roll_score;

$,=' ';
print @scores,"\n";
```

extract the values in a hash using **values** function.
The function returns the values as an array.

output

```
85  70  95  81  90
```

Check if a key in a hash exists or not

```
%roll_score = ( 15020, 70,  15220, 85, 15136, 90,  15200, 95,  15174, 81 );
$,=' ';
print %roll_score,"\n";

if ( exists($roll_score{15021}) )
{
        print "key 15021 exists";
}
else  {
        print "key 15021 does not exist";
}
```

Use the function **exists()** to find if a key exists in the hash or not.
exists() returns TRUE if the key exists.

output

```
15220 85 15020 70 15200 95 15174 81 15136 90

key 15020 exists
```

Using foreach

- foreach is useful to process elements of an array or a hash.
  - In case of hash, foreach works properly only if keys are not added or deleted inside foreach block

```
@nums=(10, 12, 15, 20, 17 );
$sum=0;
foreach $ele (@nums)
{
        $sum += $ele;
}

print "$sum\n";
```

```
%roll_score = ( 15020 => 70, 15220 => 85,
                                        15136 => 90, 15142
=> 82,
                                        15200 => 95, 15174
=> 81 );
$hiscore=0;
$rollno=0;
foreach $roll (keys %roll_score)
{
        if ($roll_score{$roll} > $hiscore)
        {
                $hiscore = $roll_score{$roll};
                $rollno = $roll;
        }
}
print "$rollno  $hiscore\n";
```

Using each function

- Each function is more efficient if the hash has a large number of entries.

```
%roll_score = ( 15020 => 70, 15220 => 85,
                                        15136 => 90,
15142 => 82,
                                        15200 => 95,
15174 => 81 );
$hiscore=0;
$rollno=0;
while (($roll,$score) = each (%roll_score))
{
        if ($score > $hiscore)
        {
                $hiscore = $score;
                $rollno = $roll;
        }
}
print "$rollno   $hiscore\n";
```

Comparison operators

- Comparison operators for numbers are similar to those in C or Java.

```
3 < 7                    # returns TRUE
2 == 2                   # returns TRUE
2.0 == 2          # returns TRUE
```

- Comparison operators for strings

gt, ge, lt, le, ne

```
'2'  gt  '3'             # returns FALSE,
                                 # in sorted order 2 comes
before 3
'abc'  lt  'def'     # returns TRUE

'abc' == 'def'     # does numeric comparison because
                                 # operator used is ==
```

Command-line arguments

- Perl scripts can accepts commandline arguments.
- commandline arguments received by to perl script are accessible using ARGV array.
- ARGV array contains all arguments passed on the script.
- ARGV[0] represents the first argument and so on.
- Unlike argv in C/C++, ARGV does not contain the command itself.
- n-th argument can be accessed using the scalar $ARGV[n], where n is an integer

```
#file : cmdarg.pl
print $ARGV[0]. "   " . $ARGV[1]."\n";
```

```
$ perl  cmdarg.pl  7  15
7    15
$  perl  cmdarg.pl  xy  z
xy  z
$
```

Command-line arguments (Contd.).

- Perl has no equivalent of argc as in C/C++.
- Count of arguments can be obtained using $#ARGV.
- The following program prints each argument passed on to the script.

```
#file : args.pl
$args = $#ARGV + 1;
print "no. of args = $args\n";
for ($k=0; $k < $args; $k++)
{
   print "ARGV[$k] = $ARGV[$k]\n";
}
```

```
$ perl  args.pl  a  b  c
no. of args = 3
ARGV[0] = a
ARGV[1] = b
ARGV[2] = c
$ perl  arg.pl  a  b  c  d  e
no. of args = 5
ARGV[0] = a
ARGV[1] = b
ARGV[2] = c
ARGV[3] = d
ARGV[4] = e
$
```

Example

```
#file : sum.pl
$sum;
$argc=$#ARGV + 1;
for ( $k=0, $sum=0; $k < $argc; $k++)
{
    $sum += $ARGV[$k];
}
print  $sum . "\n";
```

```
$ perl  sum.pl  1  2  3  4  5
15
$
$ perl  sum.pl  10  20  30  40
100
$
```

Branching and Looping

Agenda

- Branching constructs
- Looping constructs
- Breaking from a loop
- Jumping statements

Branching Constructs

- Branching constructs in Perl are
  - if (expression) { S }
  - if (expression) { S1 } else { S2 }
  - if (expression) { S1 } elsif (expression) { S2 }
  - if (expression) { S1 } elsif (expression) { S2 } else { S3 }
  - unless (expression) { S }
    - This is equivalent to

```
if (expression) { } else { S }
```

if-else and if-elsif-else constructs

```
# read a number
# from stdin & print
# if it is even or odd
my $num;
$num = <STDIN>;
chomp($num);
print "$num is ";
if ( $num % 2 == 0)
{
  print "even\n";
}
else
{
  print "odd\n";
}
```

```
my $num;
$num = <STDIN>;
chomp($num);
print "$num is ";
if ( $num == 0)
{
        print "zero\n";
}
elsif ( $num > 0 )   {
        print "+ve\n";
}
else   {
        print "-ve\n";
}
```

unless construct

```
$x=1;
if ($x > 0 )
{
   print "x is positive\n";
}
unless ($x == 0)
{
   print "x is not  0\n";
}
```

Looping Constructs

- Looping constructs in Perl include
  - for
  - foreach
  - while (expression) { S }
  - until (expression) { S }
  - do { S } while (expression)
  - do { S } until (expression)

Variants of for loop

```
for  $i  (1 .. 10)
{
   print "$i\n";
}
```

```
@array = <stdin>;
print "------\n";
print @array;
print "------\n";
for  $i  ( @array )
{
  print  $i;
}
```

```
for $i  ( "1 2 3 4" )
{
      printf("$i\n");
}

print "---------\n";

for $i  ( 1, 2, 3, 4 )
{
      printf("$i\n");
}
```

```
for ($x=0 ; $x < 20; ++$x)
{
              if ( $x % 2 ) {
        printf("$x\n");
              }
}
```

foreach loop

- **foreach** repeats a loop for each value of a list specified.

```
foreach  $k  (1, 2, 3, 4)
{
   print "$k \n";
}

@nums=(1, 2, 3, 4);
foreach   $k  (@nums)
{
     print "$k \n";
}
```

```
$nums="1,2,3,4";

foreach   $k  ($nums)
{
   print "$k \n";
}

$nums="1 2 3 4";
foreach   $k  ($nums)
{
   print "$k \n";
}
```

foreach – next

- **next** statement causes execution of the current iteration to skip the remaining part of the iteration and continue to the next iteration.

```
@list = (1, 3, 5, 6, 0, 2, 0, 3, 7, 0, 12 );

foreach $k ( @list )
{
  if( $k == 0 )
  {
      next;
  }
  print  "$k ";
}
```

Output

```
1  3  5  6  2  3  7  12
```

foreach – last

- **last** statement is used to break a loop.
- last can have label name specified.

```
@list = (1, 3, 5, 6, 0, 2, 3 );

LABEL:
{
   foreach   $k ( @list )
  {
        if( $k == 0 )   {   last LABEL;   }
        print  "$k ";
   }
   print "after  foreach loop\n";
}
print "after LABEL\n";
```

Output

```
1  3  5  6  after  LABEL
```

```
@list = (1, 3, 5, 6, 0, 2, 3);

foreach   $k ( @list )
{
   if( $k == 0 )   {   last;   }
   print  "$k ";
}
print  "\n";
```

Output

```
1 3 5 6
```

while & until loop

- Note that the same loop when represented by until has the condition reversed in comparison to while loop.

```
$x=0;
while ($x < 10)
{
    if ( $x % 2 )
  {
        printf("$x\n");
    }
    ++$x;
}
```

Output

```
1 3 5 7 9
```

```
$x=0;
until ($x > 10)
{
  if ( $x % 2 )
  {
        printf("$x\n");
  }
  ++$x;
}
```

Output

```
1 3 5 7 9
```

Jumping statements

- **goto** statement lets control jump to the point labeled in the code.
- **redo** lets a block be reexecuted.
- A redo statement can have a condition clause **rep if**, in which case reexecution of the block is done only if the condition is true.

```
$x=1;

rep:        # define goto label

   print   $x   . "\n";
   ++$x;
   sleep 1;    # sleep for 1 sec.
   goto  rep;
```

```
$x=1;
rep:
{
   print   $x   . "\n";
   ++$x;
   redo rep if ($x < 5);
}
```

Subroutines

Agenda

- At the end of this module you will know:
  - How to define subroutines
  - Invoking subroutines with or without arguments
  - The purpose of @_special variable
  - How to return values from subroutines

Subroutines – Introduction

- Subroutine is a function that isolates specific logic from rest of the program
- Advantages of subroutine
  - Make a Perl program well structured
  - Subroutines are reusable
  - Subroutines that take arguments make them usable in more than one context
  - Splitting a program's logic into multiple subroutines improves maintainability of the program

Defining & Invoking Subroutines

- A function is defined as shown below

```
sub prntmsg                    # prefix function name with the keyword sub
{
   print "prntmsg routine called\n";
}
```

- A function is invoked by calling the function name in other parts of the program

```
prntmsg;
prntmsg;
prntmsg;
```

Defining & Invoking Subroutines

```
$msg="";            # global variable

sub prntmsg
{
  print "$msg\n";# access global variable inside the function
}

prntmsg;

$msg="hello";                # set global variable and then invoke the function
prntmsg;

$msg="hello world";                    # set global variable and then invoke the
    function
prntmsg;
```

Passing Arguments

- A subroutine accesses arguments passed to it using the special variable ***$_*** which represents an array of the arguments.
  - $_[0] represents the 1st argument, $_[1] represents the 2nd argument, etc.

```
sub prntmsg
{
  print "$_[0]\n";
}
```

- Two ways to pass arguments to a function as shown below.

```
prntmsg("hello");
prntmsg "hello world";
```

@_ Special Variable

- @_ represents a list of all arguments passed to a subroutine.

```
sub  max
{
      ($a, $b) = @_;
      $maxval = $a;
      if ($b > $maxval)    {   $maxval = $b;  }
      $maxval;   # return value
}
print  max(5, 4),"\n";
print  max(-5, 4),"\n";
print  max(5, -4),"\n";
print  max(-5, -5),"\n";
```

Output

```
5
4
5
-5
```

@_ Special Variable

```
sub  max
{
     ($count, @nums) = @_;
     $maxval = $nums[0];
     for ($n = 1; $n < $count; $n++)
     {
          if ( $maxval < $nums[$n])
         {
              $maxval = $nums[$n];
         }
      }
      $maxval;   # return value
}
@nums=(3, 5, 0, 1, -2, 4, 10, -5);
print  max($#nums, @nums);
```

Assigns the first argument to $count and the remaining to @nums

Output

10

Private variables – my operator

- In Perl all variables are global by default.
- However, it is possible to have private variables (also known as lexical variables) using ***my*** operator.
- A variable marked with "my" are private to the enclosing block.
- ***my*** operator is used to mark variables inside a subroutine local to it, thus insulating a global variables.

```
$t=100;
print "outer t = $t\n";
{
   my $t=200;
   print "block-1 t = $t\n";
   {
      my $t="hello";
      print "block-2 t = $t\n";
   }
   print "block-1 t = $t\n";
}
print "outer t = $t\n";
```

Output

```
outer t = 100
block-1 t = 200
block-2 t = hello
block-1 t = 200
outer t = 100
```

Returning Values from Subroutines

- Use return statement to return a value from a subroutine.

```
sub getmax          # returns the max
{
    my $max=0;

    if ( $_[0] > $_[1] )
    {
        $max = $_[0];
    }
    else
    {
        $max = $_[1];
    }
    return $max;
}
```

```
# using the subroutine

$x=getmax(11,51);
print  $x . "\n";

$x=getmax -10,-5;
print  $x . "\n";

$x=getmax(1,1);
print  $x . "\n";
```

Returning Value from Subroutine's argument

- Use return statement to return a value from a subroutine.

```
sub getmax
    # returns the max
   # through the 3rd argument
{
   $_[2] = $_[0];

   if ( $_[1] > $_[0] )
   {
      $_[2] = $_[1];
   }
}
```

```
# using the subroutine

getmax(11,51, $x);
print  $x . "\n";

getmax -10,-5, $x;
print  $x . "\n";

getmax(1,1, $x);
print  $x . "\n";
```

Perl Scripting

Agenda

- References
- File I/O
- Packages & OOPs
- Pattern Matching

Objectives

At the end of this module you will know:

- How to use references in Perl
- The different File I/O operations using Perl
- How to implement Packages and modules
- How to write classes in Perl and use them
- Performing pattern matching

References

Objectives

At the end of this module you will know:

- How to create references for scalars, arrays, hashes and subroutines
- How to use references
- How to define anonymous Arrays, hashes and subroutines
- How to us file handlers and typeglobs

References for scalars, arrays & hashes

- Reference is a pointer to a variable or value.
- Reference is a separate data type, but is a kind of scalar value.
- Perl references are like C++ or Java references.
- It is possible to take references to scalars, arrays and hashes.
- Prefixing a variable with backslash (\) returns its reference.
- To dereference a reference, prefix it with the symbol that is used to represent the corresponding datatype.

References for scalars, arrays & hashes

Type	taking a reference	Dereferencing
Scalar	$a = 100; **$refa = \$a;**	**$$refa=200;** **print "$$refa\n";**
Array	@chars= (a .. e); **$refchars=\@chars;**	**$$refchars[0]**='A'; print **@{$refchars}**;
Hash	%char2num = ('a', 1, 'b', 2, 'c', 3); **$ref_char2num = \%char2num;**	**$$ref_char2num{'b'}**=100; print **%$ref_char2num**,"\n";

Example – reference to scalar

```
$a = 100;
$refa = \$a;
print  "\$a = $a\n";
print  "\$refa = $refa\n";
print  "\$\$refa = $$refa\n";
$$refa=200;
print  "\$a = $a\n";
print  "\$refa = $refa\n";
print  '$$refa = ' . "$$refa\n";
```

output

```
$a = 100
$refa = SCALAR(0x1698f90)
$refa = 100
$a = 200
$refa = SCALAR(0x1698f90)
$$refa = 200
```

Example – reference to scalar

```
$x = hello;
$refx = \$x;
print  "\$x = $x\n";
print  "\$refx = $refx\n";
print  '$$refx = ' . "$$refx\n";
$$refx=longtext;
print  "\$x = $x\n";
print  "\$refx = $refx\n";
print  "\$\$refx = $$refx\n";
```

output

```
$x = hello
$refx = SCALAR(0x23e30f0)
$$refx = hello
$x = longtext
$refx = SCALAR(0x23e30f0)
$$refx = longtext
```

Example – reference to array

```
@chars= ( a .. e );
$,=' ';
print @chars;
print "\n";
$refchars=\@chars; # take reference to array
print @{$refchars}; # print array using ref.
print "\n";

# modify array elements using reference
$$refchars[0]='A';
$$refchars[1]='B';
$$refchars[5]='F';
print @{$refchars}; print "\n";
print @chars; print "\n";
```

output

```
a b c d e
a b c d e
A B c d e F
A B c d e F
```

Example - Array of references

```
@rgb=("red", "green", "blue");
@othercolors=("black", "white", "orange");
@arycolors=(\@rgb,\@othercolors);
$,=' | ';
print @{$arycolors[0]};
print "\n";

$arycolors[0][0]="RED";
print @{$arycolors[0]};
print "\n";

print @{$arycolors[1]}; print "\n";
$arycolors[1][4]="yellow";
print @{$arycolors[1]};  print "\n";
```

output

```
red | green | blue
RED | green | blue
black | white | orange
black | white | orange |  | yellow
```

Example – reference to hash

```
%country_cap = ( 'France', 'Paris', 'India', 'New Delhi',
                                    'UK', 'London', 'US', 'Washington');
$country_cap = \%country_cap;      #get reference to hash
@country = keys %$country_cap;
@caps = values %$country_cap;
$,=' - ';
print @country;  print "\n";
print @caps;  print "\n";
```

output

```
UK - France - India - US
London - Paris - New Delhi - Washington
```

Anonymous array

- Anonymous array is an unnamed array
  - created by enclosing array elements in square brackets.
  - It can only be accessed using a reference to it, which must be stored in a scalar variable, when the anonymous array is created.

```
my @ary = qw(p q r);
print "@ary\n";
#define anonymous array and store its reference
my $ary_ref = ["a","b","c"];
print "@$ary_ref\n";
push (@$ary_ref, "d");
print "after push -- @$ary_ref\n\n";
my $ary_ref2 = [qw(x  y  z)];    #one more anon. array
print "@$ary_ref2\n";
pop(@$ary_ref2);
print "after pop -- @$ary_ref2\n";
```

output

```
p q r
a b c
after push -- a b c d
x y z
after pop -- x y
```

Anonymous hash

- Anonymous hash is an unnamed hash
  - created by enclosing hash elements in curly brackets.
  - It can only be accessed using a reference to it, which must be stored in a scalar variable, when the anonymous hash is created.

```
$ref_country_cap = { 'France' => 'Paris', 'India' => 'New Delhi',
                     'UK' => 'London', 'US' => 'Washington' };

my @country = keys %$ref_country_cap;
my @caps = values %$ref_country_cap;
$,=' - ';
print @country; print "\n";
print @caps;  print "\n";
```

output

```
UK - France - India - US
London - Paris - New Delhi - Washington
```

File I/O

Objectives

At the end of this module you will know:

- How to open a file for file I/O
- Use of die and warn function
- How to perform different operation on files
- Use of file test operators
- How to delete a file

Opening & closing a file

- Perl script can open a file using **open** operator.
- A file can be opened for read/write/append operations.
- File handle name is specified as the first argument, which will have the file handle if the file open is successful.
- The second argument specifies which file to be opened.
- The filename can be optionally prefixed with < or > or >> symbol.

```
open  FILEHANDLENAME, 'filename'          # open for reading

open  FILEHANDLENAME, ' <filename'        # open for reading

open  FILEHANDLENAME, ' >filename'        # open for writing

open  FILEHANDLENAME, ' >>filename'       # open for append
```

Note: File handles STDIN, STDOUT and STDERR are pre-opened.

Opening & closing a file

- Following is also another way to open a file

```
open FILEHANDLENAME, 'filename' ;          # open for reading
open FILEHANDLENAME, '<', 'filename' ;     # open for reading
open FILEHANDLENAME, '>', 'filename' ;     # open for writing
open FILEHANDLENAME, '>>', 'filename' ;    # open for appending
```

To close a file use the **close** operator.

```
close FILEHANDLENAME;
```

Reading & writing to files

- Read file from standard and write to standard output.

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

- Read from disk file given as argument and write to standard output.

```
open(FILE, "<", $ARGV[0]);

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

- Read from a file and write to another file

```
# copy source file to destination file

open(SRCFILE, "<", $ARGV[0]);
open(DESTFILE, ">", $ARGV[1]);

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

Handling file open error

- In case an application can't continue logically if a file can't be successfully opened, provision exists to show an error message and then terminate the application.
  - **die** operator can be used for the purpose as shown below.

```
open(SRCFILE, "<", $ARGV[0]) or die "$0: Can't open file $ARGV[0] - $!\n";
open(DESTFILE, ">", $ARGV[1]) or die "$0: Can't open file $ARGV[1] - $!\n";
while(<SRCFILE>)   {
   print DESTFILE   "$_";
}
close  SRCFILE;  close  DESTFILE;
```

Note:

- $0 represents the command executed
- $! Represents the system defined error message

Handling file open error

- If an application can continue logically, even if a file can't be successfully opened, application can prompt a message using **warn** operator
  - **warn** operator prompts the message, but lets program to continue execution after that.

```
my $k=0;
for ($k=0; $k <= $#ARGV; $k++)
{
   open(DATA, "<", $ARGV[$k])  or
          warn "$0: Can't open file $ARGV[$k] : $!\n";

     if (DATA == 0)
     {
           while(<DATA>)
           {
                print "$_";
           }
     }
     close DATA;
}
```

File I/O – handling binary files

- To read a file in binary mode, file should be opened as **raw** file, as shown in the example below.
- File should be read using the read call.

```
open(SRCFILE, "<:raw", $ARGV[0]) or die "$0: Can't open file $ARGV[0] - $!\n";
open(DESTFILE, ">:raw", $ARGV[1]) or die "$0: Can't open file $ARGV[1] - $!\n";

my $buf;            # declare a buffer

while(read(SRCFILE, $buf, 1024))   # read 1024 bytes into buf
{
   print DESTFILE   $buf;          #  write buffer contents to destination file
}

close  SRCFILE;
close  DESTFILE;
```

Reading entire file into an array

- It is possible to read entire file into an array, as shown the example given below.

```
open(FILE, $ARGV[0]);
my  @lines = <FILE>;     # reads entire file into the array line
close(FILE);
print  @lines;
```

Data as part of perl script

- Perl lets data to be part of the script file itself and use the normal file I/O mechanism to read the data.
- The special directive __DATA__ indicates beginning of the data.
- Lines after this represent the data.
- This data is read using the handle DATA .

```
while (<DATA>)    #  read one line
{
        print $_;        # print the line read
}
__DATA__
one   two   three
four five
six seven  eight
```

output

```
one   two   three
four five
six seven eight
```

File test operators

- Perl has a variety of operators that can be used to test attributes of a file.
- Some commonly used file test operators are

- r $file	→ true if $file is readable
-w $file	→ true if $file is writable
-e $file	→ true if $file exists
-z $file	→ true if file exists and is empty
-s $file	→ true if file exists and is not empty

- Some operators that test the file type are

- f $file	→ true if $file is plain text file
-d $file	→ true if $file is directory
-l $file	→ true if $file is a symbolic link
-p $file	→ true if file is a named pipe

File test operators – example

```
# filename: getfiletype.pl

foreach ( @ARGV )
{
    print "$_  - ";

    if (-f $_) { print "file\n"; }

    if (-f $_) { print "ordinary file\n"; }

    elsif (-p $_) { print "named pipe\n"; }

    else { print "type unknown\n"; }

}
```

```
$ perl getfiletype.pl    /etc/ /etc/passwd /bin/ls

/etc/ - directory

/etc/passwd - ordinary file

/bin/ls - ordinary file

$
```

Deleting a file

- ***unlink*** function can be used to delete a file.
- If the operation is successful, it returns *true*; else returns *false*.

- The below code snippet deletes files specified as arguments to the perl script.

```
foreach ( @ARGV )
{
        unlink  $_  or  warn "Can't delete  $_\n";
}
```

Packages & OOPs

Packages

- Packages is the foundation, on which modules and classes are built.
- Packages resemble to namespaces in C++
- Packages minimize the collision of variable names between packages.

- A package introduces new scope, which is effective till another package is defined.
- A file can have multiple packages.

- Reusable units in Perl are in the form of **modules**.

Declaring Packages & accessing members

```
$val=5;        # val in global namespace
print $val . "\n";

package pkgA;    # introducing new namespace pkgA
$val=100 + $::val;     # val local to pkgA
print $val . "\n";

package pkgB;    # introducing new namespace pkgB
$val=200;
print $val . "\n";

package main; # introducing new namespace pkgB

$val = $pkgA::val + $pkgB::val +  $::val;

print   $val . "\n";

print   $::val . "\n";
```

Output

```
5
105
200
310
310
```

val in pkgB namespace

val in global namespace

Declaring Packages

- Packages can be sequenced in any order, as long as it does result in uninitialized values.

```
$val=5;
print $val . "\n";

package pkgA;
$val=100;
print $val . "\n";

package pkgB;
$val=200;
print $val . "\n";

package main;

$val = $pkgA::val + $pkgB::val +  $::val;
print $val . "\n";
```

Output

```
5
5
100
200
```

Package variable vs Lexical variable

- Package variables belong to the package in which they are defined.
- Lexical variables do not belong to any package.
  - They are declared using ***my*** keyword.
- Lexical variables are local to the code block or file in which they are declared.

Perl Module

- A Perl program can contain several packages, some of which can be reusable by some other Perl programs.
- A package defined inside a Perl program file, limits its usage.
- For better reusability a package has to be maintained as a separate file.
- Such a file is known as Perl module and the file is given the extension **.pm**
- Perl Modules facilitate
  - Code reusability
  - Code sharing
  - Code maintenance (bug fixes, addition of features)

Perl Module

- Perl has a huge collection of modules, which can be found at CPAN (Comprehensive Perl Archive Network) www.cpan.org .
- These modules can be classified as
  - Pragmatic modules
  - Standard modules
  - Extension modules

Perl Module

- A perl module can contain only one package.
- Name of the perl module is the name of the package.

```
#filename: sample.pm

package sample;


sub  PrintHello1
{
    print "Hello-1\n";
}

sub  PrintHello2
{
    print "Hello-2\n";
}

1;
```

Package name

Indicates end of module.
This is the value returned by the module.
Module should return true (1) indicating success.

Using a Perl Module

- The use keyword is used to identify the module to be included which contains the relevant package.
- For example to use the sample package which is places in sample.pm
- Perl locates the module using the built-in variable INC with contains a list of searchable paths for modules.
- Note that processing of use is done at compile time, not at run time.

```
use sample;

sample::PrintHello1;

sample::PrintHello2;

PrintHello1;

PrintHello2;
```

Using a Perl Module

```
#filename: sample.pm
use strict;
use warnings;

package sample;

require Exporter;
our @ISA="Exporter";
our @EXPORT = qw(PrintHello1);

sub  PrintHello1
{
    print "Hello-1\n";
}

sub  PrintHello2
{
    print "Hello-2\n";
}

1;
```

```
use strict;

use warnings;

use sample  qw(PrintHello1
                    PrintHello2);

PrintHello1;

PrintHello2;
```

Output

```
Hello-1
Hello-2
```

OOPs Concepts of Packages

- Packages is the foundation, on which modules and classes are built.
- Packages resemble to namespaces in C++
- Packages minimize the collision of variable names between packages.

- A package introduces new scope, which is effective till another package is defined.
- A file can have multiple packages.

- Reusable unit in Perl are in the form of **modules**.

Objects in Perl

- In Perl an object is a reference.
- A reference can be converted into an object using *bless* operator.
- Since object has attributes, it is represented using a hash.
- So, an object is reference to a hash.

Defining Date class

```
#filename: Date.pm

use strict;
use warnings;

package Date;   # indicates beginning of the package named "Date"

sub new    {                    # this is like C++/Java constructor to create a new object
      my $class = shift;          # name of the class
      my $self = { };             # create a hash reference
      $self->{day} = shift;
      $self->{mth} = shift;       # initialize members with values from constructor call
      $self->{year} = shift;

      bless ($self);  # associate the reference with the class Date. Now object is made
      return $self;   # return the Date object
}


1;
```

Creating instances of Date class

```
use Date;    # use the Date module

my $dt1 = Date->new(1,3,2012);   # create one Date object

print $dt1, "\n";

my $dt2 = Date->new(1,3,2012);    # create another Date object

print $dt2, "\n";
```

Output

```
Date=HASH(0x15d8170)
Date=HASH(0x15d8284)
```

Date class with methods

```
#filename: Date.pm

use strict;
use warnings;

package Date;

sub new {
	my $class = shift;
	my $self = { };
	$self->{day} = shift;
	$self->{mth} = shift;
	$self->{year} = shift;

	bless ($self);
	return $self;
}
```

```
sub getday   {
	my $self = shift;
	return $self->{day};
}

sub getmth   {
	my $self = shift;
	return $self->{mth};
}

sub getyear   {
	my $self = shift;
	return $self->{year};
}

1;
```

Testing Date class with methods

```
use Date;

sub printdate
{
	$dt = shift;

	my $d = $dt->getday();
	my $m = $dt->getmth();
	my $y = $dt->getyear();

	print "$d-$m-$y\n";
}

my $dt1 = Date->new(1,3,2012);

printdate $dt1;

my $dt2 = Date->new(15,10,2013);

printdate $dt2;
```

Output

```
1-3-2012
15-10-2013
```

Pattern Matching

Objectives

At the end of this module you will know:

- Special operators, Quantifiers, The character class, Special meta characters
- Anchors, Modifiers, Special variables
- Back references, Substitution
- Translation
- Split function
- Join function

Pattern Matching

- Unix has a number of utilities that help a program or user to search for a pattern of text in files and print the lines that match the criteria.

- Some of the Unix utilities for pattern matching include
  - grep
  - awk
  - sed
  - … etc.

- Perl is much more versatile and is powerful in pattern search.

Quantifiers

- A quantifier identifies the number of times a pattern can be matched consecutively.

- Following are the quantifiers that can be used in regular expressions.

Quantifier	Meaning
*	match 0 or more times
?	match 0 or 1 time
+	match at least once
{m}	match exactly *m* times
{m,}	match *m* or more times
{m,n}	match at least *m* times or at most *n* times

Quantifiers – example

```
#file : quantifier.pl

sub  PrintMatchedLines
{
        my $pat = shift;

        while (<STDIN>)
        {
                if ( /$pat/ ) { print; }     # print line, if pattern is found
        }
}

PrintMatchedLines  ($ARGV[0]);
```

Quantifiers – example (Contd.)

data.txt

```
abc
aabbcc
aabbbc
abbbcccc
aaaabc
aaabbbccc
abbbc
aabbbb
ac
cacacb
ababac
abacab
abacac
abcabc
```

```
$ perl quantifier.pl abc < data.txt
abc
aaaabc
abcabc
$
$ perl quantifier.pl a+bc < data.txt
abc
aaaabc
abcabc
$
$ perl quantifier.pl "a{2,3}bc" < data.txt
aaaabc
$
$ perl quantifier.pl (abc){2} < data.txt
abcabc
$
```

Quantifiers – example (Contd.)

data.txt

```
abc
aabbcc
aabbbc
abbbcccc
aaaabc
aaabbbccc
abbbc
aabbbb
ac
cacacb
ababac
abacab
abacac
abcabc
```

```
$ perl quantifier.pl "a{2,}b{2}c" < data.txt
aabbcc
$
$ perl quantifier.pl "(ab){2,}" < data.txt
ababac
$
$ perl quantifier.pl "^a{2,}(a|b|c)*c$" < data.txt
aabbcc
aabbbc
aaaabc
aaabbbccc
$
$ perl quantifier.pl "^.{7,}$" < data.txt
abbbcccc
aaabbbccc
$
```

Character class

- A character class represents a set of characters that are similar type.
- Some of the character sets are listed below.

Character set	Class name		Escape sequence	
Upper case characters	[:upper:]	[^[:upper:]]	\u	\U
Lower case characters	[:lower:]	[^[:lower:]]	\l	\L
All letters	[:alpha:]	[^[:alpha:]]	\a	\A
Digits	[:digit:]	[^[:digit:]]	\d	\D
Blank characters	[:space:]	[^[:space:]]	\s	\S
Printable characters	[:print:]	[^[:print:]]	\p	\P

Special meta-characters

- Meta characters have special meaning given by Perl.
- If they are part of a pattern to be matched, they must be prefixed with escape character (\ backslash)

Meta character	**Example**	**Remarks**
$	“$abc”	Matches with a string that beginnings with **abc**
^	“abc^”	Matches with a string that ends with **abc**
[ ]	[abc]123	a123, b123, c123, but not d123
()	([ab])([bx]+)	Matches with a character sequence that begins with either **a** or **b** and followed by either **b** or **x** one or more times. Parenthesizing is necessary for backreferencing

Anchors

- Anchor is a special character that indicates whether the patter should be found at the beginning of the line or end of the line.
- Perl supports two anchors **^** and **$**
- **^** - this appears at the beginning of a pattern string. e.g., "^Total"
  - The pattern "^Total" matches with lines that begin with **Total**
- **$** - this appears at the end of a pattern string. e.g., "bash$"
  - The pattern "bash$" matches with lines that end with **Bash**
- It is possible for a patter string to have both anchors
  - For example, the pattern "^Total$" matches those lines that has **Total** as the only text.
  - The pattern "^$" matches with empty lines.

Note: These anchors have interpretation in **grep** utility.

Anchors

```
$ ls -l /usr | perl -e 'while (<>) { if (/^d/) { print $_;} }'
dr-xr-xr-x.   2 root root 36864 Aug 25  2013 bin
drwxr-xr-x.   2 root root  4096 Jun 28  2011 etc
drwxr-xr-x.  34 root root  4096 Feb  5 10:52 include
...
dr-xr-xr-x.   2 root root 12288 Aug 20  2013 sbin
drwxr-xr-x. 205 root root  4096 Aug 20  2013 share
drwxr-xr-x.   4 root root  4096 Apr 15  2013 src
$
```

```
$ perl -e 'while (<>) { if (/nologin$/) { print $_; } }' /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
...
$
```

Modifiers

- Modifier is a character specified at the end of a regular expression to modify the behaviour.
  - **/i ignore case.** By default pattern matching is case sensitive.
  - **/m** modifier lets a string to be treated as multiple lines.
    Useful, if the string has multi-line text
    (i.e., \n character embedded in the string.)

```
my $text;
my $files="cat.h\ncount.c\ncat.c\nstring.cpp\nCat\n";
$text=$files;
$text =~ s/^cat/CAT/g;
print $text . "-------\n";
$text=$files;
$text =~ s/^cat/CAT/gm;
print $text . "-------\n";
$text=$files;
$text =~ s/^cat/CAT/igm;
print $text . "-------\n";
```

```
CAT.h
count.c
cat.c
string.cpp
Cat
-------
CAT.h
count.c
CAT.c
string.cpp
Cat
-------
CAT.h
count.c
CAT.c
string.cpp
CAT
-------
```

Example – working in unix

```
$ cat data.txt
one two
from to
fromday today
from
to
$ perl -p  -e 's/from/to/g' data.txt
one two
to to
today today
to
to
$
```

Example – working in unix

```
#file : pat.pl
$pat = shift;
while (<>)
 {
      if (/$pat/)
      {
            print $_;
       }
}
```

```
$ perl pat.pl "^while" < pat.pl
while (<>) {
$ perl pat.pl  "}"  < pat.pl
}
}
$ perl pat.pl  "^if"  < pat.pl
if (/$pat/) {
$ perl pat.pl  "pat"  < pat.pl
#file : pat.pl
$pat = shift;
if (/$pat/) {
$
```

Special Variables

- Perl has special variables that hold back references, that get filled when a m// or s/// matches are made.
  - $1, $2, $3, … hold the matched pattern.
  - $+ holds the highest backreference
  - $& holds the entire regex that matches

- Perl provides special variables that indicate start & end positions of the matched pattern

  $-[0] - start of the entire regex match

  $-[1], $-[2] etc, - start of the 1st, 2nd … matched patterns

Substitution

- Substitution function to replace a pattern with another text is similar to the way provided by ***vi*** editor.
- Substitution function is in the following form

$string =~ **s**/pattern/replacementtext/

If the string in which substitution has to be done is $_, then
The following is sufficient

s/pattern/replacementtext/

Substitution (Contd.).

```
# regex-subst.pl
# read from STDIN
while (<>)
{
    $line = $_;
    $line =~ s/:x:/::/;
    print $line;
}
```

Data.txt

```
one:X:two:x:three:x:
four:x:five
six:X:seven:x:eight:X:
nine:x:ten:x:
eleven:X:twelve
```

```
$ perl  regex-subst.pl  <  data.txt
one:X:two::three:x:
four::five
six:X:seven::eight:X:
nine::ten:x:
eleven:X:twelve
$
```

Note:
only the first matching occurrence is replaced in each line

```
# regex-subst.pl
# same functionality as above, but cryptic

while (<>)
{
    s/:x:/::/;      #substitute in the string $_
    print;          # print $_
}
```

Substitution – g & i modifiers

- Use **g** modifier for all occurrences to get replaced.

```
# regex-subst.pl
# read from STDIN
while (<>) {
    s/:x:/::/g;
    print;
}
```

```
one:X:two:x:three:x:
four:x:five
six:X:seven:x:eight:X:
nine:x:ten:x:
eleven:X:twelve
```

```
$ perl regex-subst.pl < data.txt
one:X:two::three::
four::five
six:X:seven::eight:X:
nine::ten::
eleven:X:twelve
$
```

- Use **i** modifier to perform case in-sensitive pattern match

```
# regex-subst.pl
# read from STDIN
while (<>) {
    s/:x:/::/gi;
    print;
}
```

```
one:X:two:x:three:x:
four:x:five
six:X:seven:x:eight:X:
nine:x:ten:x:
eleven:X:twelve
```

```
$ perl regex-subst.pl < data.txt
one::two::three::
four::five
six::seven::eight::
nine::ten::
eleven::twelve
$
```

Transliteration

- Perl provides transliteration operator **tr**.
- The transliteration operator is bound to the text with =~.
- Transliteration transforms input text based on the character mapping

- **tr** operator usage is in the following format

$*string* =~ **tr** /*fromsequence*/*tosequence*/

This transforms the input string such that those characters of the *fromsequence that appear in the string* are replaced by corresponding characters given in the *tosequence*

Transliteration - Example

```
$text="this is a sample line of text";

$text =~ tr/ /-/;            # replace space with hyphen

print  "$text\n";

$text =~ tr/a-z/A-Z/;        # replace lower case with upper case

print  "$text\n";

$text =~ tr/-//d;            #  use delete modifier to delete character(s)

print  "$text\n";
```

Output

```
this-is-a-sample-line-of-text
THIS-IS-A-SAMPLE-LINE-OF-TEXT
THISISASAMPLELINEOFTEXT
```

Transliteration – Example (Contd.).

- if the *fromsequence* has more characters than the *toseqeunce*, all characters of the *fromsequence* that have no corresponding character in the *tosequence* are mapped to the last character of the *tosequence*.

```
$text="Hello! Good morning. How are you?";

$text  =~  tr/!.?/ /;

print "$text\n";

$text =~ tr/ /\n/;

print   "$text\n";
```

Output

```
Hello  Good morning  How are you
Hello

Good
morning

How
are
you
```

Split function

- Split operator breaks a string into multiple components based on a delimiter or a pattern.
- Split returns a list of substrings

split /regex/, string

```
$curdate="01-jan-2013";
@date = split /-/, $curdate;
print "$curdate\n";
print "day   → $date[0]\n";
print "month → $date[1]\n";
print "year  → $date[2]\n";
```

Output

```
01-jan-2013
day → 01
month → jan
year →  2013
```

Split function - example

- Program to print the loginid, user id and home directory of each user.

```
open(FILE, "< /etc/passwd");
while(<FILE>) {
@fields = split(/:/, $_);
printf("%-10s  %5d  %-20s\n",
  $fields[0], $fields[2], $fields[6]);
}
close(FILE);
```

Output

```
root        0  /bin/bash
bin         1  /sbin/nologin
daemon      2  /sbin/nologin
adm         3  /sbin/nologin
lp          4  /sbin/nologin
sync        5  /bin/sync
shutdown    6  /sbin/shutdown
...
```

Join function

- Join takes an array and joins elements of the array delimiting the elements by a separator as specified by the first argument.

join (separator, array);

```
@mth_30_days = qw(apr  jun  sep  nov);

print  join(',', @mth_30_days );  print "\n";

print  join(' - ', @mth_30_days );  print "\n";

print  join('', @mth_30_days );  print "\n";

print  join("\n", @mth_30_days );  print "\n";
```

Output

```
apr,jun,sep,nov

apr - jun - sep - nov

aprjunsepnov

apr

jun

sep

nov
```

Perl Scripting

- CPAN
- Perl DBI
- Message Logging

Objectives

At the end of this module you will know:

- What is CPAN and its importance
- How to interact with databases using Perl
- Message Logging facilities in Perl

CPAN

Objectives

At the end of this module you will know:

- What is CPAN and its importance
- How to browse/search for modules in CPAN
- How to use ActivePerl PPM to download Perl Modules

CPAN (Contd.).

- CPAN (Comprehensive Perl Archive Network – www.cpan.org) is a repository of redistributable, and reusable Perl modules
- CPAN has been in existence since 1993
  - Has over a hundred thousand Perl modules in its collection.
  - More than ten thousand contributors
  - Has mirrors sites and search facilities (search.cpan.org or metacpan.org)
  - One of the largest archives of libraries
- Perl modules on CPAN are released under open-source license
- Perl programmers around the globe have contributed *libraries* (known as *modules* in Perl terminology) that can also used by other programmers.
  - Facilitates reusability
  - Avoid ‘reinventing the wheel’

CPAN

- Perl modules are archived in CPAN in the form of distributions.
- A distribution contains
  - One or more Perl modules
  - Installation & test scripts
  - Documentation

- A distribution is uploaded by its author to Perl Authors Upload Server (PAUSE).

CPAN Categorization

- Perl modules cover different areas , such as
  - Archiving & Compression – Archive::, Compress::, RPM::
  - Database interfaces – DBD::, DBI::,
  - Networking - IPC::, Net::
  - OS interfaces – Linux::, Proc::, Unix::
  - Security – Authen::, Crypt::
  - Text processing – PDF::, Text::, XML::
  - World Wide Web – CGI::, HTML::, HTTP::

CPAN client

```
cpan shell -- CPAN exploration and modules installation (v1.9800)
Enter 'h' for help.

cpan> _
```

CPAN client

```
cpan shell -- CPAN exploration and modules installation (v1.9800)
Enter 'h' for help.

cpan> install Net::Ping_
```

CPAN client

```
cpan shell -- CPAN exploration and modules installation (v1.9800)
Enter 'h' for help.


cpan> install Net::Ping
Database was generated on Thu, 18 Apr 2013 02:39:47 GMT
Running install for module 'Net::Ping'
Running make for S/SM/SMPETERS/Net-Ping-2.41.tar.gz
Checksum for I:\Dwimperl\cpan\sources\authors\id\S\SM\SMPETERS\Net-Ping-2.41.tar
.gz ok
Scanning cache I:\Dwimperl\cpan\build for sizes
.......................................................................DONE
```

CPAN client

```
cpan> install Net::Ping
Database was generated on Thu, 18 Apr 2013 02:39:47 GMT
Running install for module 'Net::Ping'
Running make for S/SM/SMPETERS/Net-Ping-2.41.tar.gz
Checksum for I:\Dwimperl\cpan\sources\authors\id\S\SM\SMPETERS\Net-Ping-2.41.tar
.gz ok
Scanning cache I:\Dwimperl\cpan\build for sizes
.......................................................................DONE


  CPAN.pm: Building S/SM/SMPETERS/Net-Ping-2.41.tar.gz

Checking if your kit is complete...
Looks good
Writing Makefile for Net::Ping
Writing MYMETA.yml and MYMETA.json
cp lib/Net/Ping.pm blib\lib\Net\Ping.pm
  SMPETERS/Net-Ping-2.41.tar.gz
  I:\Dwimperl\c\bin\dmake.EXE -- OK
Running make test
I:\Dwimperl\perl\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness(0, 'b
lib\lib', 'blib\arch')" t/*.t
```

CPAN client

```
t/140_stream_inst.t .. ok
t/150_syn_inst.t ..... ok
t/190_alarm.t ........ ok
t/200_ping_tcp.t ..... ok
t/250_ping_hires.t ... ok
t/300_ping_stream.t .. ok
t/400_ping_syn.t ..... ok
t/410_syn_host.t ..... ok
t/450_service.t ...... ok
t/500_ping_icmp.t .... ok
t/510_ping_udp.t ..... ok
t/520_icmp_ttl.t ..... ok
All tests successful.
Files=16, Tests=134, 64 wallclock secs ( 0.27 usr +  0.14 sys =  0.41 CPU)
Result: PASS
  SMPETERS/Net-Ping-2.41.tar.gz
  I:\Dwimperl\c\bin\dmake.EXE test -- OK
Running make install
Installing I:\Dwimperl\perl\lib\Net\Ping.pm
Appending installation info to I:\Dwimperl\perl\lib/perllocal.pod
  SMPETERS/Net-Ping-2.41.tar.gz
  I:\Dwimperl\c\bin\dmake.EXE install UNINST=1 -- OK


cpan>
```

Perl DBI

Objectives

At the end of this module you will know:

- Architecture of DBI
- How to use DBI to perform database operations such as
  - Creating a table
  - Adding records to a table
  - Performing queries
  - Deleting records
  - Deleting tables
- How to trace program execution

Perl DBI

- Perl DBI (Database Interface) is generic module that provides connectivity between Perl and any database.
  - DBI communicates with database specific database driver (DBD) module.

- DBI uses Object-Oriented features of Perl.

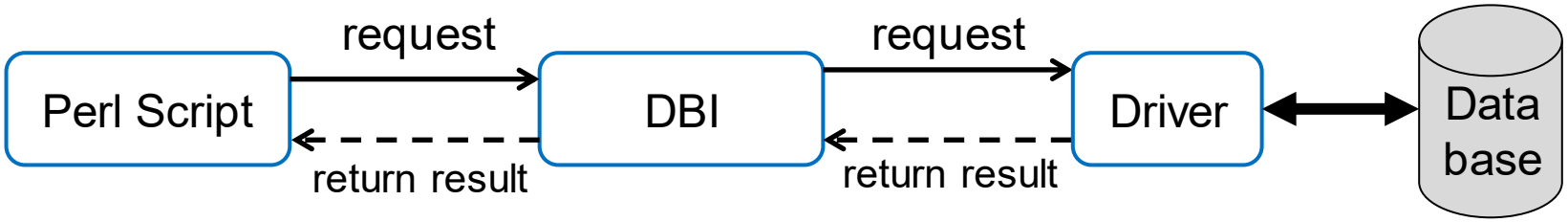
- An application that intends to operate on a database has to use DBI module, with statement

```
use DBI;
```

This results in DBI module getting loaded.

DBI Architecture

- DBI architecture comprises of two components
  - DBI
    - Provides programming interface (methods), which Perl scripts use to perform various operations with databases
    - DBI routes these methods to appropriate driver(s) depending on with which database application interacts
    - DBI mediates between Perl script and databases, forwarding request to the appropriate driver and returning the results from driver to the application
  - Drivers (also known as Database Drivers (DBDs))
    - Drivers are defined in DBD namespace
    - Drivers for most popular databases such as Oracle, Informix, DB2, MySQL etc. exist



DBI Handles

- DBI defines 3 types of objects or handles.
- Driver handle
  - Through which communication with driver takes place
  - Based on the data source specified when a connection is requested, corresponding driver is loaded
- Database handle
  - Returned by DBI when a connection with a database for a specific user is established
  - A Perl script can have multiple database handles, each representing one pair of database type and user.
- Statement handle
  - It interacts and manipulates data.
  - For example, prepare() returns statement handle.
  - A database handle can have multiple statement handles.

Data Source Name

- Data source name identifies where DBI has to find the database.
- DBI requires the data source name specified as showb below.

dbi:*database_driver_name*:*database_name*

- Data source name must start with **dbi**.
- *database_driver* name identifies which database driver to be loaded.
- *database_name* can be just name of the database, if it is a local database or fully qualified name that specifies hostname, database and port number, if it is a remote database.

Steps in Programming using DBI

- Any application to interact with a database has to use DBI module, which provides necessary classes and interfaces.

```
use DBI;
```

1) The first step to interface with a database is to connect to the database using ***connect()*** method of DBI class.

```
my $dbHandle;        # handle to database
my $dbHandle = DBI->connect( "dbi:Oracle:mydb",
                             "trng", "trng123" )
        or die "Connect() failed. " . $DBI::errstr\n";
```

Connect method returns handle to the database

Steps in Programming using DBI (Contd.).

2) Now call ***prepare()*** to prepare the database to execute a SQL statement.
   - on success *prepare()* returns statement handle.

```
my $stHandle;          # handle to statement
$stHandle = $dbHandle>prepare('SELECT * FROM   TEST')
            or die "prepare() failed" . $dbHandle->errstr;
```

3) Call ***execute()*** on the statement handle to execute the statement.
   - On success it returns true; otherwise false.

```
$stHandle->execute()
            or die "execute() failed" . $stHandle->errstr;
```

Steps in Programming using DBI (Contd.).

4) On successful execution of the query, get rows(or records) returned by the database using ***fetchrow_array().***
   - This returns the fields of one record at a time.
     - Fields of a record are returned as an array.
   - Call the function repeatedly to get all the records returned by the database.
     - If there are no more rows, the function returns an empty list.

```
my  @fields;
@fields = $stHandle->fetchrow_array();
print   $fields[0],  "     ", $fields[1];  # print 1st & 2nd fields
```

5) If no more operations on the database, call ***disconnect(),*** on the database handle.

```
$dbHandle->disconnect();
```

Putting all together

```
use DBI;
my $dbHandle = DBI->connect( "dbi:Oracle:mydb", "trng", "trng123" );

my $stHandle = $dbHandle->prepare('SELECT * FROM TEST');
$stHandle->execute();

my @fields;   # array to hold contents of a row
while ( @fields = $stHandle->fetchrow_array() )   {
        my  $isbnno = $fields[0];
        my  $title = $fields[1];
        printf  "%10s   %-20s\n", $isbnno, $title;
}

$dbHandle->disconnect();
```

output

```
1449364977  Mastering Perl
1118013840  Beginning Perl
0764537504  Perl For Dummies
0132381826  Perl by Example
```

Simplifying the use of fetchrow_array()

- It is possible to capture values of row fields into respective variables, without the need for explicit assignments, as shown by the alternative implementation.

```
my @fields;
while ( @fields = $stHandle->fetchrow_array() )   {
        my   $isbnno = $fields[0];
        my  $title = $fields[1];
        printf  "%10s   %-20s\n", $isbnno, $title;
}
```

Alternative way

```
my ($isbnno, $title);

while ( ($isbnno, $title) = $stHandle->fetchrow_array() ) {
        printf "%10s   %-20s\n",$isbnno, $title;
}
```

do() method

- DBI class has ***do()*** method, which can be used to shorten the process of making the *prepare()* followed by *execute()* sequence.
- do() method combines these two steps, and is invoked on a database handle returned by *connect()*.
- do() method takes SQL statement as a string.
- do() method is convenient to execute various SQL statements such as
  - Create
  - Insert
  - Truncate
- **Note:** When same SQL statement has to be executed multiple times, *prepare()* gives better performance than *do().*

Creating a table

```
my $dbHandle = DBI->connect( "dbi:Oracle:mydb",
                                        "trng", "trng123" );

my $sql_statement = " CREATE   TABLE     TEST (
          isbn   VARCHAR(10),
          title   VARCHAR(30)
     )";

$dbHandle->do("$sql_statement");
```

Adding records to a table

```
sub AddRow
{
	my ($handle, $isbn, $title) = @_;

	my $sql_stmt = "INSERT INTO TEST (isbn, title)
			VALUES ('$isbn', '$title')";
	$handle->do("$sql_stmt");
}

sub PopulateTable
{
	my $dbHandle = shift;

	AddRow($dbHandle, '1449364977', 'Mastering Perl');
	AddRow($dbHandle, '1118013840', 'Beginning Perl');
	AddRow($dbHandle, '0764537504', 'Perl For Dummies');
	AddRow($dbHandle, '0132381826', 'Perl by Example');
}
```

Delete All Rows of a table

```
sub DeleteAllRows
{
    my ($handle) = @_;

    $sql_stmt = "truncate table test";
    $handle->do("$sql_stmt");
}

DeleteAllRows($dbHandle);
```

Binding parameter with actual value at runtime

- Most often an SQL query is executed multiple times, but with different values, for each query.
  - select * from test where isbn = '1449364977';
  - select * from test where isbn = '0764537504';
- Databases optimizes execution of such queries, using two-step process - prepare & execute.
- As part of *prepare* step, the query is passed on, with actual values unspecified, instead filled by placeholders.

select * from test where isbn = '**?**'; ## here placeholder is '?'

- At a later stage of execution, application provides the actual values for the placeholders.
- Perl provides *bind_param()* to bind actual value.
  - Each placeholder is numbered (numbering starts from 1)

Binding parameter with actual value at runtime

```
my $sql_stmt = "select * from test where isbn = ? ";
my $stHandle = $dbHandle->prepare($sql_stmt);

$stHandle->bind_param(1, '0132381826');
$stHandle->execute();
@fields = $stHandle->fetchrow_array();

$stHandle->bind_param(1, '0764537504');
$stHandle->execute();
@fields = $stHandle->fetchrow_array();

… similar SQL queries with same/different isbn value
```

This placeholder is numbered 1

Bind value with the 1st placeholder

No need to perform prepare()

Recap

- DBI interfaces that have been used
  - connect - connects to a database
  - prepare - prepares for execution
  - execute - executes an SQL statement that was prepared
  - do - perform the combined operation of prepare & execute
  - bind_param - bind actual parameter with placeholder
  - fetchrow_array - get row returned by the database
  - disconnect - disconnect with the databas

Logging

Logging

- A logging system provides a variety of messages that can be logged by application
- Messages can be differentiated on the importance (priority) of the message, such as - INFO, WARNING, FATAL
- It is possible to filter messages based on the importance
- Logs can be targeted to standard output or standard error or to a disk file.

- CPAN has several modules for logging
  - Log::Message
  - Log::Message::Simple
  - Log::Log4Perl
  - Sys::Syslog (for UNIX)
  - Win32::Eventlog (for Windows)

Log::Log4Perl module

- Log4Perl Module is one of the highly configurable logging modules
- It supports a variety of destinations (*appenders*) for logging such as scree, file, mail etc.
- Supports 5 different levels at which messages can be logged.
  - TRACE (least priority), DEBUG, INFO, WARN, ERROR, FATAL (highest priority)
- Messages get logged if the message priority is greater or equal to the priority level chosen by the logger.
- Messages can be logged using a subroutine that has the same name or calling methods with the same name (but lower case) on the logging object.
- Log4Perl can be configured to suit application specific requirements, such as type of filtering, target for logging (specifying one or more appenders), format of logging.

Log::Log4Perl – filtering messages

```
use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init($TRACE);

TRACE   'This msg is of type - TRACE';
DEBUG   'This msg is of type - DEBUG';
INFO   'This msg is of type - INFO';
WARN    'This msg is of type - WARN';
ERROR   'This msg is of type - ERROR';
FATAL   'This msg is of type - FATAL';
```

Set the filter level

Output with the setting of easy_init($TRACE)

```
2014/03/13 14:19:49 This msg is of type - TRACE
2014/03/13 14:19:49 This msg is of type - DEBUG
2014/03/13 14:19:49 This msg is of type - INFO
2014/03/13 14:19:49 This msg is of type - WARN
2014/03/13 14:19:49 This msg is of type - ERROR
2014/03/13 14:19:49 This msg is of type - FATAL
```

Output with the setting of easy_init($INFO)

```
2014/03/13 14:19:49 This msg is of type - INFO
2014/03/13 14:19:49 This msg is of type - WARN
2014/03/13 14:19:49 This msg is of type - ERROR
2014/03/13 14:19:49 This msg is of type - FATAL
```

Log::Log4Perl – using logging object

```
use Log::Log4perl qw(:easy);
Log::Log4perl->easy_init($INFO);
my $logger = Log::Log4perl->get_logger();
$logger->trace('This msg is of type - TRACE');
$logger->debug('This msg is of type - DEBUG');
$logger->info('This msg is of type - INFO');
$logger->warn('This msg is of type - WARN');
$logger->error('This msg is of type - ERROR');
$logger->fatal('This msg is of type - FATAL');
```

Get logging object

Use the methods to log messages

```
2014/03/13 14:19:49 This msg is of type - INFO
2014/03/13 14:19:49 This msg is of type - WARN
2014/03/13 14:19:49 This msg is of type - ERROR
2014/03/13 14:19:49 This msg is of type - FATAL
```

Log::Log4Perl – Configuring the module

Config file named app-l4p.conf

```
log4perl.rootLogger=INFO, display
log4perl.appender.display=Log::Log4perl::Appender::Screen
log4perl.appender.display.layout=Log::Log4perl::Layout::SimpleLayout
```

Replace *easy_init* call in the Perl script with the following to initialize the logger

```
Log::Log4perl::init('app-l4p.conf');
```

output

```
INFO - This msg is of type - INFO
WARN - This msg is of type - WARN
ERROR - This msg is of type - ERROR
FATAL - This msg is of type - FATAL
```

Note:

- In the above configuration file, **display** is a name, which can be replaced with any other name of programmer's choice

Log::Log4Perl – directing output to a file

Config file to direct log output to a file

```
log4perl.rootLogger=INFO, myfile
log4perl.appender.myfile=Log::Log4perl::Appender::File
log4perl.appender.myfile.filename=app-logfile.txt
log4perl.appender.myfile.mode=replace
log4perl.appender.myfile.layout=Log::Log4perl::Layout::SimpleLayout
```

Note:

- myfile – any name of programmer's choice
- app-logfile.txt – any name of of programmer's choice
  this is the name of the file in which messages are logged
- Choose mode as "replace". Otherwise log messages are appended to the existing file