CS202: PROGRAMMING PARADIGMS & PRAGMATICS

Semester II, 2020 – 2021

Lab 3: Introduction to Perl

Aim: Introduce you to programming in Perl using command-line interface in Linux. *This lab assumes that you are familiar and comfortable using the Linux operating system!!*

- **Introduction:**
  - Perl means *P*ractical *E*xtraction and *R*eport *L*anguage. Perl is a programming language that is both easy to learn and highly flexible.
- **Let's get started!**
  - Create a directory structure to hold your work for this course and all the subsequent labs:
    - Suggestion: `CS202/Lab3`
- **Directions**
  - Make a file for each of the following examples. Then execute it. Make sure each program works, and that you understand it. Try making small changes to each.
  - To run a Perl Program:
    - Create a file *name.pl* with the program in the example
    - Type: perl *name.pl*
- **First Perl Program: Hello World!**
  - The first Perl program that you write will be in a file named `HelloWorld.pl`
  - To create that file and start editing it, use the following command in the terminal (file names are case-sensitive): `gedit HelloWorld.pl`
  - File will be created in the current directory, which should be `CSL202/Lab4`
  - The traditional first program just says "Hello World!". To make that happen, type the following lines of Perl code into the editor:

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

print "Hello, world!\n";
```

    - Line 0: Normally, # would indicate a comment in Perl, but here it expresses the location of the Perl interpreter.
    - Line 1: The string "Hello, world!" is printed on the screen. Notice that each statement ends with a ";"
  - Back on the command line, run the program by entering the command: `perl HelloWorld.pl`
  - If you have made no mistakes while typing the program, this program should simply print 'Hello, World!'
- **A Program with Variables:**
  - Variables are memory location to store information. In Perl, variables are type less i.e there is no data type like int, char, etc. Every variable is a string and depending on the context will be treated as int, float etc.
  - There are 4 kinds of variables namely: scalars, lists, arrays and hashes.
  - Scalar variables contain singular value like 10, hello etc. Name of scalar variable is prefixed with `$` symbol and followed by a letter and then letters, digits or underscores.

- To begin, use the command in the terminal: `gedit Variables.pl &`
- Then type the following program into the gedit window:

```
#!/usr/local/bin/perl
$Num1 = 3;
$Num2 = $Num1++;
$Pi = 3.14;
$Famous_Name = "Karen A. Lemone";
print "The numbers are: $Num1,  and $Num2,  and $Pi \n";
print "$Famous_Name \n";
$name = ”Raman”;
print “hello $name”;
print 'hello $name';
$name = ”ram”;          # in string context
$age = 30;              # in numerical context
$age = $age+1;          # treated as numeric
$age1 = $age.$age;      # treated as string.
                        # '.'(dot) is a concatenate operator
print “$age”
print “$age1”
```

  - Line 1: The integer 3 is assigned to $Num1.
  - Line 2: $Num2 is assigned the value 3; then $Num1 is incremented
  - Line 3: $Pi is assigned 3.14
  - Line 4: $Famous_Name is assigned the string "Karen A. Lemone"
  - Line 5: The numbers are printed.
  - Line 6: The string is printed.
  - Line 8: (double quote) is used when interpolation/substitution is required
  - Line 9: (single quote) is used when it is a literal string. Special characters will not be interpreted.
  - Remaining lines: Comments provide explanation

- Save your program, and run it, to make sure it works, using the commands in the terminal:

```
perl Variables.pl
```

- **An Array Example:**
  - Arrays are used to store multiple ordered values. Array variables should have prefix `@`. The size of array need not be specified beforehand. Each element of the array is scalar and can contain mixed data types. Index starts with zero.
  - Whenever the whole array is required `@array` will be used. Suppose we want only an element then `$array[ ]` will be used as every element is a scalar, thus the prefix `$.`
  - Create a new file using the command in the terminal: `gedit Arrays.pl &`
  - Then type the following program into the gedit window:

```
#!/usr/local/bin/perl
$Num1 = 3;
@Num1 = (1, "one", 1.5, $Num1);
print "The array is: @Num1 \n";
print "The number is $Num1 \n";
$Num1[0] = 5;
print "The array is: @Num1 \n";
@array = (1,2,3);      # Assignment – whole array using list
print @array;          # Operations on Array
print "@array";
$array[3] = 4;         # Assigning element
print "@array \n";
push @array,'4';       # New element can be added at the end using push
print "@array \n";
$last = pop @array;    # Last element can be removed using pop function
print "last = $last\n";
@array = (1,2,3);
$first =shift @array; # First element can be removed using shift
print "first = $first\n";
@array = (3,4,5);
unshift @array,'2';    # Element can be added at the front using unshift
print "array = @array\n";
```

- Line 1: The scalar $Num1 is assigned 3.
- Line 2: The array @Num1 is assigned the 5 values shown. $Num1[0] is assigned the number 1, $Num1[1] is assigned the string "one", etc.
- Line 3: The values of the array @Num1 are printed.
- Line 4: The value of the scalar $Num1 is printed.
- Line 5: The first entry in the array is changed to a 5.
- Line 6: The values of the array @Num1 are printed.
- Remaining lines: Comments provide explanation

- Save your program, and run it, to make sure it works, using the commands in the terminal:

```
perl Arrays.pl
```

- **Other Perl Constructs:**
  - Next sections introduce other Perl programming constructs. You don't have to create a separate file for each, but create one example file and keep adding to it.
  - After coding examples for each construct, execute it. Make sure each version works, and that you understand it. Try making small changes to each.

- To begin, use the command in the terminal: `gedit Example.pl &`

- **Lists:**
  - List variables are noted by symbol `()`. List is just a list of values – may be constants, scalars etc.

    ```
    (a,b,c) or ($name,$age,$sex)
    ```

  - They can be referred with index also. The index are specified inside a square bracket `[ ]`.

    ```
    $first = (a,b,c)[0];
    print "$first\n";        # Will output a
    ```

  - List variables can be assigned like this

    ```
    ($name,$age) = ('Raman',20);
    ```

- **Hashes**
  - Hash is associative/named array. They are key-value pairs. It is similar to array, except that we can use strings as index instead of 1..n.
  - Hash variables will have `%` as prefix. The contents of hash are called values and index is called key.

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

  - Other way of populating a hash

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

  - Here the list should contain even number of values. First element will be treated as key, second element value, third element key, fourth value and so on and so forth. In short odd elements will be keys, even elements will be values.
  - Individual elements of hash are accessed by means of *$hash{key}*

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

  - Adding new element:

    ```
    $fruits{'orange'}='orange';
    ```

    - Note the { } instead of [ ] as in the case of array;

- **Conditionals – IF**
  - The `if` statement is similar to if in C language, except
    - Flower brace is required even for single statement
    - else if is noted by `elsif` (note missing e).

    ```
    $mark = 40;
    if ($mark>75){
       print "passed with distinction\n";
    }
    elsif ($mark<35){
       print "failed\n";
    }
    ```

```
else {
   print "passed\n";
}
```

- Alternate form of if statement is

```
print "variable a is >10" if ($a>10);
```

- **Loops**
  - `for` loop syntax is similar to C. It can also be used for iterating on a list. `foreach` is same as for. Both `for` and `foreach` are used interchangeably for readability.
  - Classical `for` as in 'C':

```
for ($i=0;$i<10;$i++){
      print "i=$i\n";
}
```

  - The other way of using `for` is below:

```
foreach $i (a,b,c) {
 print uc $i;
}
```

  - Explanation: `foreach` will execute the body once for every element in the list – 3 times in this case. Each time the variable `$i` will get the value it is iterating ie. `$i` will be '`a`' first time '`b`' second time and '`c`' the third time. `uc` – is a perl function to change a string into upper case.
  - You can combine functions like '`print uc $i`' instead of `print (uc($i))`. Also note that brackets are optional for passing arguments to functions like `uc`, `print`.
    - The output will be ABC.
  - Looping on hashes – keys function

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

  - Explanation: keys is a function which return a list of key values. The list ( ) will contain apple, banana, grape while running.
  -
  - `while` loop is used to iterate and has syntax similar to C.

```
$i=0;
while ($i<10){
   print "i=$i\n";
   $i++;
}
```

- **Accepting input**
  - Keyboard inputs can be accepted using `<STDIN>`

```
print "enter your name ";
$name=<STDIN>;
```

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

- **Default scalar variable $_**
  - $_ is called default variable. It will be used if no other variable is specified.

```
foreach (a,b,c){
      print uc ;
}
```

  - The above foreach is same as before, however $i is omitted. Still Perl will output same i.e 'ABC'.
  - This is because Perl uses default variable $_ to store and expands the lines as

```
foreach $_ ( a,b,c){
      print uc $_;
}
```

  - Similarly $_ is used in the following case where '..' the generator function is used.

```
foreach (1..10){
      print ;
}
```

- **Subroutines**
  - Subroutines can be defined using sub keyword. The arguments passed will be in a default array @_

```
$v1 = 10;
$v2 = 20;
add($v1,$v2);

sub add {
      ($a,$b)=@_;
   print $a+$b;
}
```

  - This should give output 30.
  - You can return value using return statement.

- **Scope of variables**
  - By default all variables are global i.e available throughout the file.
  - You can limit scope to a block/sub by using my.

```
$v1=10; $v2=30; #v1, v2 global
$v3=30;
$v3=add( $v1,$v2 );

sub add{
   my ($i,$j)=@_;
   print "inside add sub value of i=$i j=$j\n";
   print "inside add sub value of globals v1=$v1 v2=$v2
   v3=$v3\n";
   return $i+$j;
}

print " Value of globals v1=$v1 v2=$v3\n";
print " Value of scoped variables v3=$v3\n";
print " Value of variables inside sub i=$i j=$j\n";
```

- You can limit scope to a block also

```
for (my $i=0; $i<10; $i++ ) {
     print "inside for i=$i\n";
}
print "outside for i=$i\n";
```

- **use strict**
  - In Perl you need not define variables before using. By default all variables are global.
  - However, this may lead to errors due scope conflict or errors in naming. `use strict` is a pragma which will help in avoiding it. Once `use strict` is used, every variable has to be declared with proper scope using `my`.

```
use strict;
$v1=10;$v2=20;
add($v1,$v2);

sub add {
   ($a,$b)=@_;
   print $a+$b;
}
```

  - The above code will not run and produce error. The corrected one will be like this

```
use strict;
my $v1=10;
my $v2=20;
add ( $v1,$v2 );

sub add {
   my ($a, $b)=@_;
   print $a+$b;
}
```

- **References**
  - References are address of the variable, similar (but not exactly) to pointers in C. You can take a reference by using `\`. It can be dereferenced by using `$$`

```
$a=10;
$ref_toa=\$a;
print "value of a using reference = $$ref_toa\n Value of using
directly=$a\n Reference of a= $ref_toa";
```

- **File handling**
  - File handling can be done after opening a file and getting handle similar to C.

```
open( $fh, "<", "data.txt" );
```

  - here `$fh` – file handle which is a scalar variable. Also it is common to uppercased variable like FH.
    - < – open read only
    - data.txt – name of the file. Full path name should be given it it is not in current directory
  - File reading line by line can be done like:

```
$line = <$fh>;
```

  - File writing:

```
print $fh "hello";
```

- Example Problem: Open data.txt file. Copy contents to udata.txt duly converted into upper case

```
open ( $fh, "<", "data.txt" );    #open file read only
open ($fh1,">","udata.txt");      #Open file write mode

while ( $line = <$fh> ) {         #read line by
   print "line=$line";            #display content on screen
   print $fh1 uc($line); #write upper cased content to new file
}
close($fh);
close($fh1);
```

- **In-Built Functions**
  - Use Perl documentation to read about and experiment with built-in functions like:

`chop, grep, join, splice, split, reverse, substr, length , offset, etc.`

- **Exercise 1:**
  - Write a program that gets the first argument ($ARGV[0]), opens it as a file for reading, and prints out the contents of the file. This is similar to what the cat program on Unix does. If no argument is given, then the program dies with an error message.
  - Create and save this program in a file named `FilePrint.pl`
- **Exercise 2:**
  - Write another program, similar to the above program, which reads the file in but pushes all the lines into a list, then prints the lines back out in reverse order.
  - Also get it to count the number of lines read in from the file.
  - Create and save this program in a file named `FileReversePrint.pl`
- **Exercises 3:**
  - Write a subroutine to test if the scalar input parameter represents a leap-year. The algorithm is:
    - Years divisible by 400 are leap years. Thus, 2000 and 2400 are leap years.
    - Years not divisible by 400 but divisible by 100 are NOT leap years. Thus 1900, 2100, 2200 are not leap years.
    - Year divisible by 4 but not divisible by 100 are leap years. 1996, 2004, 2008 are leap years.
  - Return an appropriate value to indicate leap or not-leap.
  - Create and save this program in a file named `LeapYear.pl`
- **Submitting your work:**
  - All source files as one tar-gzipped archive.
    - When unzipped, it should create a directory with your ID. Example: **2008CSB1001** (NO OTHER FORMAT IS ACCEPTABLE!!! Case sensitive!!!)
    - ***Negative marks if the TA has to manually change this to run his/her scripts!!***
  - Source files should include the following: (Case-Sensitive file names!!)
    - HelloWorld.pl [1 Points]
    - Variables.pl [1 Points]

- Arrays.pl [1 Points]
- Example.pl [2 Points]
- FilePrint.pl [5 Points]
- FileReversePrint.pl [5 Points]
- LeapYear.pl [5 Points]

- ***Negative marks for any problems/errors in running your programs***
- Submit/Upload it to Google Classroom.