# ITE315 Module 3 Part A - Programming in PERL: Getting Started

Athens State University

## Contents



# 1 What is Perl?

**What is Perl?**

- **PERL**: A modern scripting language developed in the late 1980s by Larry Wall
- The overall structure is procedural like C/C++ with features adapted from shell programming

**Two Overriding Language Design Guidelines**

- *TMTOWTDI*: There is More Than One Way To Do It
- Easy things should be easy and hard things should be possible

Perl has the reputation of being the "Swiss Army chainsaw of scripting languages" because there are many different ways to do and say the same thing. Its introduction at the start of the Internet era led to its widespread adoption as a scripting language for writing the server-side components of websites (a role that has been subsumed to a point by Python and server-side Javascript).

### Perl Is Interpreted

- Perl is an interpreted language
  - Contrast with a compiled languageC/C++
- **REPL**: Read-Evaluate-Print Loop
  - Read an expression from the user and parse it into a data structure in memory
  - Evaluate the instruction based on the data structure
  - Print the result of the evaluation

## 2 A First Walkthrough

### The Usual: "Hello World", in Perl

```
#!/usr/bin/perl
print "Hello world\n"
```

- Note the change in the sha-bang; this is for UNIX-like systems, adjust as required for Windows
- More C-like in syntax here

Perl supports sha-bangs just like Bash. Here we point things at the standard location of Perl on operating systems like Linux and macOS.

### Running Your Code

```
perl -e "print 'Hello World';"
perl hello.pl
chmod u+x hello.pl
./hello.pl
```

You can pass single instructions directly to Perl from the command-line using the "`-e`" option. This is useful in Bash pipelines where you just need to adjust a line of input before passing data to the next command in the stream.

For larger things, the Perl interpreter will process and interpret a file.

You can set the permissions on a Perl script to be executable and run the script directly (assuming you have the correct entry in the sha-bang).

### Quoting Text: Single vs. Double

```
#!/usb/bin/perl
print "Hello world\n";
print 'Hello world\n';
$a = 10;
print "Value of a = $a\n";
print 'Value of a = $a\n';
```

Note that Perl, like many programming languages, has the concept of *string interpolation*. References to variables in double-quoted strings in Perl are expanded with their current values.

## 2.1 Variables And Data Types

### Duck Typing

> If it walks like a duck, talks like a duck, and smells like a duck, then there is a gosh-darn good chance that you're dealing with a duck

- Perl is a loosely type language
- From a practical standpoint, that means that the interpreter determines data types from context available to it at first use

### The Three Basic Perl Data Types

- **Scalars**: Simple variables, whose name always begins with a `$`
  - Scalars can be either a number, a string, or a reference
- **Arrays**: Arrays are ordered lists of scalars that are accessed using 0-based numeric indexes.
  - Array names always begin with the "at" symbol: "@"
- **Hashes**: Hashes are unordered sets of key/value pairs accessed by using the keys as subscripts
  - Hash names always begin with the percent sign: "`%`"

### Working With Variables

```
#!/usr/bin/perl
my $answer = -2**10;
print $answer, "n";
my $answer = (-2)**10;
print "$answer \n";
my $price = -5.35;
print abs $price , "\n";
my $pi = 3.1415927;
my $degress = 45;
my $angle = sin($degrees / ($pi * 2));
print "Angle is $angle\n";

$randomNumber = rand(100);
$randomInt = int $randomNumber;
print "$randomInt \n";
$randomFrac = $randomNumber - $randomInt;
print "$randomNumber $randomInt $randomFrac\n";
```

### So... What's up with `my`?

- That's a scope identifier.
- No... not the mouthwash but an indication to the interpreter that you are adding a new variable to your local scope
- It adds an extra level of syntax checking
- And will be useful when we start using library packages

From the Perl documentation:

> A `my` declares the listed variables to be local (lexically) to the enclosing block, file, or eval. If more than one variable is listed, the list must be placed in parentheses.

### Mixing Data Types

```
#!/usr/bin/perl
my $a = 5;
$a++;         # $a is now 6; we added 1 to it.
$a += 10;     # Now it's 16; we added 10.
$a /= 2;      # And divided it by 2, so it's 8.
print "$a\n";
$a = "8";     # Note the quotes.  $a is a string.
my $b = $a + "1";    # "1" is a string too.
my $c = $a . "1";    # But $b and $c have different values!
print "$a $b $c\n"
```

- The values of `$b` and `$c` are very different? Why?

Perl converts strings to numbers transparently when required. But note that Perl doesn't overload the `+` operator for strings.

## 2.2 Arrays and Hashes

### Arrays

```
my @lottoNumbers = (1, 2, 3, 4, 5 6);
my @months = ("July", "August", September);
my $month = $months[2];
$months[2] = "Evil Doctor Month";
```

Arrays are indexed starting at zero. Note the difference between referring to an array as a single entity, `months`, and referring to an element of the same entity, `$month[0]`. The individual elements of an array are scalars!

Note what happens when we assign an array to a scalar:

### Arrays

```
#!/usr/bin/perl
use v5.10;
my @lottoNumbers = (1, 2, 3, 4, 5, 6);
my @months = ("July", "August", September);
say "@months";
my $month = $months[2];
$months[2] = "Evil Doctor Month";
say  "$month @months";
my @summerMonths = ("June", "July", "August");
my $countOfSummerMonths = @summerMonths;
print "$countOfSummerMonths @summerMonths\n";
my @autumnMonths;
my $countOfAutumnMonths = @autumnMonths;
print "$countOfAutumnMonths\n";
```

### Hashes

```
#!/usr/bin/perl
use v5.10;
my %daysInMonth = ( "July"=>31, "August"=>31, "September"=>30 );
my $daysInSept =  $daysInMonth{September};
say $daysInSept;
my @listOfMonths = keys %daysInMonth;
say "@listOfMonths";
```

## 2.3 Conditional Statements

### if Statements

```
if ($j > 100 ) { $j = 100;}

if ($j < 60)      { $grade = "F";}
elseif ($j < 69) { $grade = "D";}
elseif ($j < 79) { $grade = "C";}
elseif ($j < 89) { $grade = "B";}
else              { $grade = "A";}
```

### unless Statements

```
my $a = 20;
unless ($a < 20) {
  print "a is no less than 20\n";
}
print "value of a is $a\n";
```

In an `unless` statement, the block of code inside the statement will be evaluated when the boolean expression is `false`.

So, what are the rules for Boolean values in Perl? In Perl, the number zero, the strings `"0"` and `""`, and undefined are considered to be `false` while all other values are considered to be `true`.

### The `switch` Statement

```
#!/usr/bin/perl
use v5.10;
use Switch;
$var = 10;
@array = (10,20,30);
%hash = ('key1'=>10, 'key2'=>20);
switch($var) {
    case 10          {print "number 100\n"}
    case "a"         {print "string a"}
    case [1..10,42] {print "number in list"}
    case(\@array)    {print "number in list"}
    case(\%hash}     {print "entry in hash"}
    else             {print "default case"}
}
```

### The Rules For A `switch` Statement

- The switch statement takes a single scalar argument of any type, specified in parentheses.
- The value is followed by a block, which may contain one or more case statements followed by a block of Perl statement(s).
- A case statement takes a single scalar argument and selects the appropriate type of matching between the case argument and the current switch value.
- If the match is successful, the mandatory block associated with the case statement is executed.

### The Rules For A `switch` Statement

- A switch statement can have an optional else case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is matched.
- If a case block executes an untargeted next, control is immediately transferred to the statement after the case statement (i.e., usually another case), rather than out of the surrounding switch block.
- Not every case needs to contain a next. If no next appears, the flow of control will not fall through subsequent cases.

## 2.4 Loops

### `while` and `until` Loops

```
#!/usr/bin/perl
use v5.10;
my $a = 10;
while ($a < 20) {
    print "Value of a: $a\n";
    $a = $a + 1;
}

until ($a == 0) {
    $a--;
    print "Counting down to $a\n";
}
```

### Be Careful About Numbers vs. Strings

```
my $yesNo = "no";
if ($yesNo = "yes") {
   print "you said yes";
}
$a = 5;
if ($a == " 5 ") {print "Numeric equality!\n";}
if ($a eq " 5 ") {print "String equality!\n";}
```

Remember that Perl automagically converts strings to numbers when it thinks it should do the conversion. The "==" operator implies that you are working with numbers, so Perl converts the value of string to the number zero and the literal string to zero. Urrrp! Here you want to use the Perl `eq` operator to do the test.

### `for` Loops

```
use v5.10;
for my $i (1,2,3,4,5) { say $i; }

for @oneToTen = (1 .. 10);

for $topLimit = 25;
for my $i (@oneToTen, 15, 20 .. $topLimit) {
    say $i;
}

%monthHas = ("July"=>31,"August"=>31,"September"=>30);
for my $i (keys %monthHas) {
    say "$i has $monthHas{$i} days.";
}

for my $marx ('groucho', 'harpo', 'zeppo', 'karl') {
    say "$marx is my favorite Marx Brother.";
}
```

The first loop will print the numbers 1 through 5 on separate lines. This demonstrates an explicit loop list. We can specify ranges using the *range* operator. So `(1,2,3,4,5)` can be written as `(1..5)`. You can also use arrays and scalars in loop lists.

# 3 Files And Strings

### Fancy Things To Do To Strings

```
#!/usr/bin/perl
my $greeting = "You are in Dr. Lewis's World of Eeeevvvvilll Now!\n";
print substr($greeting, 0, 7);
print substr($greeting, 7);

print substr($greeting, -5,4);

substr($greeting, 0, 3) = " We";

substr($greeting, 0, 0) = "Oh No! ";
```

### Fancy Things To Do To Strings

```
#!/usr/bin/perl
my @words = split(/ /, $greeting);
@words = split(/ /, $greeting, 2);

@words = ("This", "Course", "Is", "Easy");
my $statement = join(' ', @words);
$statement = join(' and ', @words);
```

The `split()` function takes two parameters: a regular expression (Remember those from when we talked about Bash?) and a scalar. You can optionally include the number of things to put into the resulting array.

If we can split, then we can certainly join things together. The `join()` function takes two parameters: a separator and an array to join.

### Reading From A File

```
#!/usr/bin/perl
use v5.10;
open my $logfile, 'log.txt' or die "Can't open log.txt: !";
my $title = "<$logfile>";
print "Report title: $title";
print while <$logfile>;
close $logfile;
```

There's a lot in this code we need to unpack. Let's start with `open`. This works with the OS to open a file. As we don't specify a mode, it is assumed that we are working in read mode.

The `or die` clause on the open is an exception handler. If we neglected to include this clause and the file didn't exist, then the application would abnormally exit with the standard system exit code. In this case, we write an indicator of what happened. One should always include an exception handler on a file open.

Remember that we said at the beginning of the lecture that Perl is designed with the thought that you can more than one way to do things? The `while` clause is a classic example of how this works. What you see in the slide is shorthand for:

```
while <$logfile> {
  print "<$logfile>"
}
```

The "<>" notation is a shorthand for the "readline" function.

### Writing To A File

```
#!/usr/bin/perl
# Open a file in overwrite mode.
open my $overwrite, '>', 'overwrite.txt' or die "error trying to overwrite $!";

# Open a file in append mode
open my $append, '>>', 'append.txt' or die "error trying to append: $!";

say $overwrite 'Here is the next content';
print $append "We're adding to the end here.\n", "And here too!"

close $overwrite
close $append
```

Now we look at writing to files. Files can be opened in either overwrite or append mode. Once opened, you use `say` and `print` to write to the files.

Of course, do remember to be polite and close the files when you're done.

# 4 Subroutines

### Subroutines

```
#!/usr/bin/perl
use 5.010;
sub multiply {
    my (@ops) = @_;
    return $opsp[0] * $ops[1];
}

for my $i (1 .. 10) {
    say "$i squared is ", multiply($i, $i);
}
```

Here we define a function `multiply()`. In Perl, parameters passed to a subroutine are stored in the special variable "`$_`", which is an array (Why?). Note what this means for the number of parameters that you can pass into a function (and checking for correctness!).

Let's revisit the `my` keyword. In this case, the variable ops is forced by the `my` keyword to be local to the function. This is important because by default everything in a Perl program is, by default, considered to be ***GLOBAL***.

Note that we can also be finer-grained in our function by assigning multiple variables declared with a `my` in a single statement. Consider how this would change the definition of our `multiply()` function:

```
sub multiply(){
  my ($left, $right) = @_;
  return $left * $right;
}
```