



B336 Internet Systems Programming

Introduction to Perl Programming

(Week 2 Lecture 2)



Lecture Objectives

- Introduction to the Perl programming language
 - Assumption: you know how to program already.



Lecture Objectives

- Relevance to unit objectives:
 - Learning objective 2: Writing software

- Relevance to assessments:
 - Most of your programming in this unit will be in the Perl programming language
 - labs
 - assignments 1 and 2



Lecture Outline

- Why use Perl for this unit?
- Perl basic language features
- Perl object-oriented features
- How to get up to speed with Perl



Introduction to Perl

- This lecture will **NOT** be a general “Introduction to Programming” lecture.
 - You should already be familiar with programming from previous units like B102.
- The purpose of this lecture is to point out special features of Perl
 - To start you on the road in using the language.
 - Please read Unit Reader 3 for a more complete introduction to the language’s syntax.
 - You will need to refer to language references at the end of this lecture when you start doing your programming exercises.



Advantages of Perl

- The main advantages for using Perl are:
 - Rapid development - can have very terse code.
 - There are literally hundreds of ways to do any one thing - programmers develop their own unique (most efficient?) way of doing things.
 - Powerful text handling facilities.
 - Hash as a fundamental data type.



Disadvantages of Perl

- Unfortunately, some of its biggest strengths are also its weaknesses
 - Because it is terse
 - Code can be completely undecipherable by non-Perl programmers
 - Less experienced programmers tend to write really bad code.
 - Because there are so many ways to do the same thing, no one can agree on the standard way
 - Hard in team work situations.
 - Perl programmers may have concentrated on text-handling at the expense, other more complex data-type development.



Why Perl in this Unit?

- Arguments about the technical strengths and weaknesses of a programming language are always endless and never conclusive.
 - So don't ever try justifying a language choice based on language features alone.
- The REAL reasons for using Perl in this unit are:
 - String manipulation features (for manipulating HTTP message text, and XML documents)
 - Importance of Perl in some development community (especially open-source and in Linux)
 - Another alternative language to expand your repertoire.



A Basic Perl Script

```
#!/usr/bin/perl  
print "Hello!" ;
```

Indicates where the perl interpreter can be found. This “shebang” line must be at the beginning of every script.

A statement to print something to the the standard output.



Comments and Statements

- When a # symbol is encountered, anything from the # to the end of the line is ignored (with the exception of the first line).
 - To comment multiple lines, use a # on each line.
- All statements in Perl end with a semicolon.
- There is no separate “main” function in Perl.
 - The statements starting from the second line gets executed from top to bottom, unless they are functions/subroutines.



Variables

- The 3 main forms of variables in Perl are:
 - Scalars - single values
 - Arrays - a list of values
 - Hashes - collection of values indexed using “keys”
- The “my” keyword indicates that a variable is a local variable.



Scalar Variables

- A scalar variable is represented by a name with the \$ symbol in front of it.
- A scalar variable can hold a single string and or a number.
- You do not have to declare whether a scalar is a string or a number.
- Eg.

```
$num = 21 ;  
$str = 'a string' ;  
my $var = 21 + 48 ;  
my $var = 'a' . 'string' ;
```



Some Common Operations

Arithmetic operators:

```
$a = 1 + 2;      # Add 1 and 2 and store in $a
$a = 3 - 4;      # Subtract 4 from 3 and store in $a
$a = 5 * 6;      # Multiply 5 and 6
$a = 7 / 8;      # Divide 7 by 8 to give 0.875
$a = 9 ** 10;    # Nine to the power of 10
$a = 5 % 2;      # Remainder of 5 divided by 2
$a++;           # Return $a and then increment it
$a--;           # Return $a and then decrement it
```

String Operations:

```
$a = $b . $c;   # Concatenate $b and $c
$a = $b x $c;   # $b repeated $c times
```

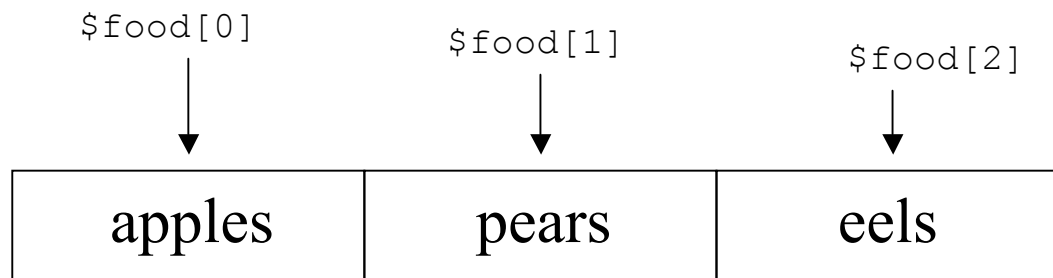
Assignments:

```
$a = $b;        # Assign $b to $a
$a += $b;       # Add $b to $a
$a -= $b;       # Subtract $b from $a
$a .= $b;       # Append $b onto $a
```

Array Variables

- Arrays are lists of scalars (ie numbers and strings).
- An array name have an @ symbol in front of it.
- Eg.

```
@food = ("apples", "pears", "eels");  
my @numbers = (1,2,3,4) ;
```





The confusion between \$ and @

- Note that in the previous example we use `$food[0]` instead of `@food[0]`.
- This is because the element is a scalar.
- Remember: use \$ when referring to the scalar elements of an array.



Some Common Array Operations

```
push(@food, "eggs");           # add "eggs" to the end of the array @food.
push(@food, "eggs", "lard");   # add 2 more to the end of the array @food.
$grub = pop(@food);           # take the last element out of array, and
                               # assign to $grub

$f = @food;                   # assigns the length of @food
$f = "@food";                 # turns the list into a string with a space
                               # between each element.

($a, $b) = ($c, $d);          # Same as $a=$c; $b=$d;
($a, $b) = @food;             # $a and $b are the first two items of @food.
($a, @somefood) = @food;      # $a is the first item of @food
                               # @somefood is a list of the others.
(@somefood, $a) = @food;      # @somefood is @food and
                               # $a is undefined.

print @food;                  # display the list.
```




Hashes

- Perl also allows us to create arrays which are accessed by a string rather than a number. These are called *hashes* (or *associative arrays*).
- A hash is prefixed by a % symbol.
- Eg.

```
%ages = ("Michael Caine" => 39,  
        "Dirty Den" => 34,  
        "Angie" => 27,  
        "Willy" => "21 in dog years",  
        "The Queen Mother" => 108);
```



Hashes

- From the previous example:

```
$ages{"Michael Caine"} returns 39
```

```
$ages{"Dirty Den"} returns 34
```

```
$ages{"Angie"} returns 27
```

```
$ages{"Willy"} returns "21 in dog years"
```

```
$ages{"The Queen Mother"} returns 108
```

- Note again the use of the symbol \$ instead of % to access the elements.



Hashes

- All the strings used to do the indexing are called *keys*.
- The function `keys` can be used to get all the keys in a hash. Eg.

```
keys(%ages)
```

gives

```
("Michael Caine", "Dirty Den", "Angie",  
 "Willy", "The Queen Mother")
```



Control Structures

- Like most other languages, Perl has control structures for conditionals and looping
- Examples of common ones:
 - if...elsif...else
 - foreach
 - for
 - while
 - do...while



if statements

- For executing statements based on a condition
- Eg.

```
if ($a == $b) {  
    print "The numbers are equal\n";  
} else {  
    print " The numbers are not equal\n";  
}
```



Test Operators

```
$a == $b      # Is $a numerically equal to $b?  
              # Beware: Don't use the = operator.  
$a != $b      # Is $a numerically unequal to $b?  
$a eq $b      # Is $a string-equal to $b?  
$a ne $b      # Is $a string-unequal to $b?  
  
($a && $b)    # Is $a and $b true?  
($a || $b)    # Is either $a or $b true?  
!($a)        # is $a false?
```



elseif

- If more than one condition:

```
if (!$a) {
    print "The string is empty\n";
} elseif (length($a) == 1) {           # If above fails, try this
    print "The string has one character\n";
} elseif (length($a) == 2) {         # If that fails, try this
    print "The string has two characters\n";
} else {                               # Now, everything has failed
    print "The string has lots of characters\n";
}
```

- Note that “elseif” is NOT spelled “elseif”!



foreach loop

- To go through each line of an array or other list-like structure (such as lines in a file)
- Eg.

```
foreach $yum (@food) {      # Visit each item in turn and call it $yum
    print "$yum\n";
}
```

```
foreach $name (keys %ages) {
    print "$name $ages{$name}\n";
}
```




Curly brackets

- Unlike most programming languages, even when your control block has only one statement (examples previous page), you **MUST** still put the curly brackets {...}.



for loops

- General form:

```
for (initialise; test; inc) {  
    first_action;  
    second_action;  
    etc  
}
```

- Eg.

```
for ($i = 0; $i < 10; ++$i)           # Start with $i = 1  
{                                       # Do it while $i  
    print "$i\n";                       # Increment $i before repeating  
}
```



Subroutines

- In Perl, we define encapsulated blocks of code by using **subroutines**
 - We sometimes call them **functions** in Perl as well.
- You may place subroutines anywhere in the code.
 - But we usually have the “main” statements at the beginning, and place all subroutines at the end.



Example Subroutines

```
#!/usr/bin/perl

&mysubroutine;    # Call the subroutine

sub mysubroutine {
    print "Not a very interesting routine\n";
    print "This does the same thing every time\n";
}
```

```
#!/usr/bin/perl

print &add(1,2); # Call the subroutine with parameters and
                # print the result

sub add {
    my ($a, $b) = @_ ; # read in the parameters into $a and $b
    return $a + $b ;
}
```



File Handling

- The simplest way to do file handling is to open a file and read the whole content into an array as strings. We can then process that array.
- Eg. Reading from a file

```
open(INFO, 'passwd.txt');           # Open the file
@lines = <INFO>;                    # Read it into an array
close(INFO);                        # Close the file
...
```



Other Example File Handling

```
open(INFO, 'passwd.txt');           # Open the file
while ($lines = <INFO>) {          # Read one line at a time
    ...                             # Note: $line and not @lines
}
close(INFO);                       # Close the file
```

```
$file = 'data.txt' ;
open(INFO, ">$file");               # Open for output
print INFO "This line goes to the file.\n"; # Overwrite the file
```

```
$file = 'data.txt' ;
open(INFO, ">>$file");               # Open for appending
print INFO "This line goes to the file.\n"; # Append to the end of the
                                           # file
```



Brackets or no-brackets

- There are two basic ways passing parameters to standard functions.
- The following are equivalent:

```
print INFO "This is a line" ;  
print(INFO "This is a line") ;
```

```
split /:/, $info ;  
split(/:/, $info) ;
```

- Note that there is a comma in the no-bracket version of `split`, but not in `print`. Lesson: check the function syntax - don't just remove all commas when you remove the brackets.



String Processing in Perl

- One of the most powerful features of Perl is its ability to processing text strings.
- It was originally created to do do string extraction and processing of text documents.
 - The word “Perl” is an acronym for “Practical Extraction and Reporting Language”.
- The power of the string processing is tied to its use of *regular expressions* (REs).



More Regular Expressions

- It is the special characters appearing in REs that gives the string processing its power (and also confusion).

- Some special characters used in REs:

.	# Any single character except a newline
^	# The beginning of the line or string
\$	# The end of the line or string
*	# Zero or more of the last character
+	# One or more of the last character
?	# Zero or one of the last character



More Regular Expressions

- Some example simple matches:

```
t.e      # t followed by anything followed by e
         # This will match      the
         #                       tre
         #                       tle
         # but not              te
         #                       tale
^f       # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters
         # This will match      un
         #                       und
         #                       undd (etc)
.*      # Any string without a newline. This is because
         # the . matches anything except a newline and
         # the * means zero or more of these.
^$      # A line with nothing in it.
```



Special Characters

- Some general special characters used in strings:

```
\n      # A newline
\t      # A tab
\s      # Any whitespace character: space, tab, newline, etc
```

- Example usage:

```
print "Hello!\n" ;           # print the word "Hello!" followed by a new line
```

```
if ($sentence =~ /\t/) {           # if there is a tab in $sentence
    print "There a tab in the sentence!\n" ;
}
```



Substitutions

- As well as matching regular expressions, Perl can make substitutions based on those matches.
- Eg. to replace occurrences of “london” with “London” in \$sentence:

```
$sentence =~ s/london/London/ ; # replace first occurrence
$sentence =~ s/london/London/g ; # replace all occurrences
                                # - global substitution
$sentence =~ s/london/London/i ; # ignore case of "london"
                                # Means will replace "lOnDon",
                                # "LONDON", "lONdon", etc.
```



Translation

- The `tr` function does a character-by-character translation.
- Eg. to replace 'a' with 'e', 'b' with 'f', and 'c' with 'g' in `$sentence`:

```
$sentence =~ tr/abc/efg/ ;
```

- Eg. to replace all lower case with upper case in `$sentence`:

```
$sentence =~ tr/a-z/A-Z/ ;
```



Splitting a string

- A very useful function for doing string processing in Perl is the `split` function.
- Eg.

```
$info = "January:February:March";  
@months = split /:/, $info ;
```

@months now has the array ("January", "February", "March")

```
$info = "January February March";  
foreach $month ( split /\s/, $info ) {  
    ...  
}
```

Same as above; "\s" means blank space.



The `$_` special variable

- One of the potential confusion aspects of Perl syntax is that in many situations in Perl, functions defaults to using the `$_` variable if no other variable is specified.
- Eg.

```
split(/&/) ;           # is the same as split(/&/,$_);
print ;               # is the same as print $_ ;
foreach (@food) ...   # is the same as foreach $_ (@food) ...
if (/blah/) ...       # is the same as if ($_ =~ /blah/) ...
```



Packages in Perl

- A ***package*** in Perl is a set of data and operations that belongs together.
 - Object-oriented (OO) programming in Perl is done using packages.
- To access anything within a certain package (or in a module), we do two things:
 - Load the package using **use** or **require**.
 - Use the double colon to refer to package.



Packages in Perl

- Eg.

```
use MyPackage ;  
$x = MyPackage::my_subroutine ;  
$y = MyPackage::$my_variable ;
```



Accessing Elements in Packages

- Sometimes you may see multiple levels of dereferencing:

```
MyPackage::ASubPackage::mysubroutine ;
```

- A **module** is basically a package where the file name is the same as the package name (eg. the module MyPackage is stored in MyPackage.pm) .
- A **library** is a more generic term which could mean a package, a module, a set of modules, etc.



Classes and OO in Perl

- The predefined classes in Perl are stored as packages.
- So, to access a class in one of those packages, we use something like:

```
MyPackage::MyClass
```



Syntax for OO in Perl

- Perl object-oriented features are built up from packages and subroutines.
 - Classes basically comes in packages.
 - A method is basically a subroutine in a package.



Creating Objects in Perl

- To create a new object from a class, we use:

```
$new_obj = MyPackage::MyClass->new ;
```

or

```
$new_obj = MyPackage::MyClass->new (init_data =>  
                                     "value") ;
```

Assign newly
created object
to variable

Call the constructor
method for the class

Set a data member of the
class to a value



Invoking Methods of Objects

- To get an object to perform one of its methods, we use something like:

```
MyPackage::MyClass->my_method(param1 => "value");
```



Further Reading

- This lecture do NOT cover the Perl language comprehensively.
- Required Reading
 - Unit Reader 3: Programming in Perl 5
- Language References
 - In the documentation that comes with your Perl installation.
 - <http://www.perldoc.com/>



Leverage your existing programming skills

- To get up to speed with Perl is:
 - Be familiar with the Perl documentation.
 - Remember: use \$ instead of @ or % for accessing array elements.
 - Use hashes for creative indexing instead of boring arrays with numerical index.
 - Understand special variables like \$_ and @_.
 - Look at lots of examples of regular expressions.