COMP421
Unix Environment for Programmers
Lecture 09: Perl Introduction

Jeff Wiegley, Ph.D.
Computer Science

jeffw@csun.edu

09/12/2005

"Any sufficiently advanced technology is indistinguishable from a Perl script."
–Programming Perl, 2nd edition

"there is a need to encrypt perl programs???"
–Stefan Schmiedl

Practical Extraction and Report Language (Pathologically Eclectic Rubbish Lister)

- A powerful replacement for AWK
- Tackles problems that are too complicated or big for shell programming.
- Handles common text manipulation and consumption tasks easily.

  For instance: To read in the data from all the files listed as command line arguments and remove all end-of-line characters.

  – `chomp(@allinput=<>);`
- Like a shell script, it's interpreted at run time and the script begins with `#!/usr/bin/perl -w`

  The `-w` is useful for causing warnings to be printed during certain conditions such as attempting to access an undefined variable.
- It is common to name perl files with a `.pl` extension but any filename will do.

Continued:

- Commens begin with a # and continue to the end of line.
- No continuation character is necessary since Perl commands are terminated by a semicolon as in C and Java.
- All variables are global[a]
- Variables are dynamic. They do not need to be declared prior to use.[b]
- Garbage collection occurs for data.[c]

---

[a] Unless they are declared with the my or local keywords.

[b] This can lead to trouble with typos.

[c] The programmer should take some care to undefine data that is no longer used so they arrays that are in use don't consume all of memory.

Scalar data:

The basic "type" of data held by a variable is called *scalar* data and consists of only two things:

1. A string. Ranging from 0 bytes to filling up memory. Comes in two flavors:
   (a) Double quoted strings:
      - Are variable interpolated (variables are replaced with current value).
      - Many escape sequences exist such as '\n'.
   (b) Single quoted strings:
      - Are not variable interpolated
      - Only two escape sequences: '\'' and '\\'
2. A number. Stored as a double precision floating point value. Perl has no concept of "integers"[a]

scalar variable names must begin with '`$`' as in '`$foo`'.

[a] Not exactly correct for advanced Perl programmers.

String operators:

Strings can be concatenated by the string concatenation operator '`.`'.

- `$thecat = "hello"."there";`

results in the string "`hellothere`".

Strings can copied/multiplied with the string multiplier operator '`x`'.

- `$many = "hello" x (3+1);`

results in the string "`hellohellohellohello`".

Strings are automatically converted to numbers as necessary.

- `$strange = 56 + "42.3 is the answer.";`

results in `$strange` holding the value 98.3[a]

[a] conversion silently truncates and ignores non-numeric data in the string.

The $_ variable:

The automatic variable $_ is very handy.

Many functions set, or operate on, this variable when no other argument has been given.

Care needs to be taken in its use because it's global[a] so it can easily get set to something else before you're done using it in the present context.

[a] unless declared local with the `local` keyword

Chop and Chomp:

Some quick examples of handy functions:

- `chop` is used to delete the last character of a string.

  ```
  $teststr = "hello world\n";
  chop($teststr);
  ```

  The `$teststr` gets the end of line character removed from it.

- `chomp` is like `chop` except that the last character is deleted only if it's an end of line character.

  ```
  $teststr = "hello world";
  chop($teststr);
  ```

  The `$teststrgets` remains unchanged.

Lists/Arrays:

Another data "type" Perl uses are *lists*

A list is ordered scalar data. For example: `("fred", "barney", 1, 45)`

An array is a variable which holds a list.

```
@array = ("fred", "barney", 1, 45)
```

As you can see, array variable names start with '`@`' as in '`@goo`'.

The scalar and array namespaces are independent. `$foo` is a completely different variable from `@foo`.[a]

You can use lists as both rvalues and lvalues...

```
($a, $b) = (@somearray));
```

scalar variables `$a` and `$b` will be assigned the first two elements of the `@somearray` array.

[a] hashes, filehandles and formats all have their own namespace as well.

Slices:

list usage can get really funky:

- Access to a particular element of an array uses a strange syntax:

  `$item = $array[4]` (retrieves the fifth element.)

  Notice that syntax involves `$` and not `@`!

- Negative array indexes work from the end of the array!

  `$item = $array[-3]` (retrieves the third last element)

- Can be used to return multiple values from a function:

  `($a, $b) = multireturnfunction($foo));`

- You can obtain "slices" from arrays:

  `($a, $b) = ($somearray[3,7]));`

  `$a` and `$b` are set to the 4^{th} and and 8^{th} array elements respectively.

- you can build lists to do funky stuff...

  `($a, $b) = ($b, $a);`

  Swaps the two elements.

Scalar vs List context

Perl can tell from context whether or not something is operating in scalar or list context.

```
$length = @somearray;
```

`$length` expects a scalar value; not a list. So, perl returns the length of the array instead of an arbitrary element from the array.[a]

Many of Perl's behaviors are controlled by scalar and list context.

[a] Very different from: `($length) = @somearray;` which would have plucked out the first element.

<. . .> operator:

Files I/O is represented by FILE HANDLES. file handles have their own namespace (so "`$variable`" is a scalar variable, but "`variable`" is a file handle; because it doesn't start with `$`.)

By traditional convention file handle names are usually capitalized. So "`variable`" would normally be "`VARIABLE`" instead.

basic File operations:

Open a file for reading:

```
open FILEH "<input.txt";
```

Open a file for writing:

```
open FILEH ">input.txt";
```

Open a file for appending:

```
open FILEH ">>input.txt";
```

Start a command and write to its standard input:

```
open FILEH "| sed -e 's,foo,buu,g'";
```

Run a command and read the command's stdout as input:

```
open FILEH "cat file.txt|";
```

Close a file: `close FILEH;`

Print a string to a particular filehandle:

```
print FILEH, "Desired string\n";
```

<. . .> operator:

The most simple way to read data from a file handle is to use the <> operator.

Read a single line from the file opened as VARIABLE:

```
$aline = <VARIABLE>;
```

When used in list context however, the entire file is read and returned as an array of lines:

```
@everything = <VARIABLE>;
```

Even cooler: `chomp(@everything = <VARIABLE>;)`

Simplified <>

When used without a file handle the `<>` operator changes its behavior:

1. When command line arguments are absent then `<>` ≡ `<STDIN>`.
2. If command line arguments were given `<>` processes each command line arguments as a name of a file to slurp input from. Thus in the example `./perl.pl -v hello` `<>` would first return all the lines from the file named `-v` and then all the lines from the file named `hello`.

This can be rather confusing. But basically `<>` iterates over all the elements of the `@ARGV` array and treats each element as a file to obtain input from.

If you want to mix command line switches and file arguments for a Perl program that uses `<>` then you need to do the following:

- Go through all the arguments first and process any command line switches (usually things that start with "`-`" or "`--`".
- Then you have delete from the @ARGV array all the command line switches leaving only filenames to process.

Conditional Branching:

`if` statements are just like *C* (and similar to Java) except the `{...}` braces are mandatory.

```
if ( ... ) {
  ...
} elsif {
  ...
} else {
  ...
}
```

(Notice that "elsif" is not a typo.)

But wait there's more than one way to do it...

“Quicky” conditions:

You can also use english language-type constrcution...

```
$a = $b unless ($c == 100);
```

```
$a = $b if ($c == 100);
```

And you can also compound statements:

`$a=$b && $c=$d` (`$c=$d` will only execute if `$b` had a non-zero value.)

```
FILE=open("<name") || die "could not find file";
```

Such expressions are evaluated left to right and only until a truth value as been determined.

Loops (While):

Again, same as *C* but braces are mandatory:

```
while ( ... ) {
  ...
}
```

There are three statements that control loop execution:

1. `last` *label* is used to terminate the loop labeled *label* (Or the inner-most loop if label is omitted same as `break;` would in *C*.)
2. `next` *label* terminates only the current iteration of the loop labeled *label* (Or the inner-most loop if label is omitted, same as `continue;` would in *C*.)
3. `redo` *label* Restarts at the beginning of the loop labeled *label* without evaluating the loop conditional. (Or the inner-most loop if label is omitted. There is no equivalent in *C*.)

More loops:

Perl's motto is "There's more than one way...".

```
unless ( ... ) {
  ...
}
```

```
for ( ... ; ... ; ... ) {
  ...
};
```

```
do {
  ...
} while ( ... );
```

```
do {
  ...
} until ( ... );
```

Iterating over elements of an array:

Iteration over array elements is common task. Perl has a loop for exactly that:

```
foreach $variable ( @somelist ) {
  ...
};
```

If the variable name is ommitted then foreach will assign elements to the variable `$_`.

“Quicky loops”:

It’s just a crazy, useful, free language.[a]

```
$b++ until ($b==4);
```

```
$b-- while ($b>0);
```

It has the advantage of being able to do it “your” way.

The disadvantage of of a lack of structure, often leading to incomprehensable code.

[a] sort of a “hippie” language if you will (except for the “useful” part.)

Hashes:

One great feature of Perl is built in hash tables: Like `$` and `@` for identifying scalar and array namespaces, `%` is used to identify another namespace for hashes.

`%hoo` is a hash named "hoo".

`$hoo{"key"} = "value"` assigns a value to a key.

basically, hashes are method for providing a one way association of keys to values; like is used in Java's TreeMap class.

There are a couple of functions for hashes:

- `keys(%somehash)` returns a list of all the keys stored in the table in no particular order. Since the mapping is one directional this list is guaranteed to be unique (no two keys are the same.)
- `values(%somehash)` returns a list of all the values associated with keys in the hash in no particular order. (This list may contain duplicates.)

Making complicated datastructures rapidly:

Datastructures are used to organize data.

How well you organize data directly effects how easily you can solve a problem.

Perl allows Hashes or Hashes of Arrays of Hashes or any other combination.

So for instance you can have a statement such as:

```
%myhash{$akey}[6]{$anotherkey}=$filename}
```

1. `{$akey}` will return some value that is actually an array.
2. `[6]` will return the seventh element of that array.
3. `{$anotherkey}` will treat that item as a hash and assign the value in `$filename` to the key `$anotherkey`.

It takes some getting use to but compounding Perl's basic types like this can lead to very powerful organization without the need to program new data structures yourself. But it can be a bit confusing to read.

Regular Expressions:

Perl is really good at manipulating text. This is largely due to its inclusion of regular expressions as part of its syntax.

Making complicated datastructures rapidly:

Making complicated datastructures rapidly:

Making complicated datastructures rapidly:

Making complicated datastructures rapidly:

Making complicated datastructures rapidly:

Making complicated datastructures rapidly:

Making complicated datastructures rapidly: