Introduction to Perl

(including Perl 5)

Phil Spector

Statistical Computing Facility
Department of Statistics
University of California, Berkeley

What is Perl?

- Practical Extraction and Report Language
- Developed by Larry Wall in the late 1980s
- Combines features of `awk`, `sed`, `grep` and shell scripts
- Originally developed for UNIX; now available widely
- No inherent limitations to the size of problems it can handle

Some Typical Uses of Perl

- **NOT** for computational algorithms
- preprocessing input for other programs
- postprocessing output from other programs
- repetitive editing, searching or processing of a series of files
- simplified access to system calls
- managing tasks requiring coordination among programs
- emerging as a language of choice for WWW CGI scripts
- prototyping more ambitious projects

Accessing Perl

Under UNIX, create an executable file whose first line is

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

and whose remaining lines contain your perl program

Alternatively, invoke perl from the command line:

`perl` *`optional_flags perl_program ...`*

Some of the flags available include:

`-e` - invoke perl with a one line program
`-w` - invoke perl in warning mode
`-n` - invoke perl with automatic input loop
`-d` - invoke perl with symbolic debugger

Flags can be combined. A useful example is

```
perl -de 0
```

which will run the debugger with no program.

Variables

- All variables begin with a special symbol:
  `$` - scalar   `@` - array   `%` - associative array
- Each type of variable has its own namespace
- Variable names are case sensitive: `$var`, `$Var` and `$VAR` are three different variables
- By default, all variables are global
- Perl determines type (numeric or character, scalar or array) by context
- Numbers initialize to 0, characters initialize to `""`
- Eliminate variables with `undef`

System Variables

There are many built-in variables which control the behavior of perl; each of the variables has a terse name, as well as a one or more mnemonic ones. To access the long names in the table below, include the line “`use English;`” in your `perl` program

Name	Long Name	Name	Long Name
`$/`	`$INPUT_RECORD_SEPARATOR`	`$\`	`$OUTPUT_RECORD_SEPARATOR`
`$,`	`$OUTPUT_FIELD_SEPARATOR`	`$.`	`$INPUT_LINE_NUMBER`
`$<`	`$UID`	`$>`	`$EUID`
`$$`	`$PROCESS_ID`	`$&`	`$MATCH`
`$!`	`$OS_ERROR`	`$?`	`$CHILD_ERROR`
`$|`	`$OUTPUT_AUTOFLUSH`	`$^T`	`$BASETIME`
`$ARGV`	name of file being read	`@ARGV`	array of command line args

Arrays

- Regular arrays (`@`) are indexed by number - negative numbers count from the end of the array
- The highest index of an array can be found using `$#array`
- By default, indexing of arrays starts at 0
  Can be changed with automatic variable `$[`
- Associative arrays (`%`) are indexed by strings
- Associative arrays use curly braces (like `$ary{$value}`)
- Subscripting can be used to create scalars or arrays
  Suppose `@T` is an array with 10 elements

```
@part = @T[0,1,2];  # array with three elements
$part = $T[0];      # scalar
```

- More complex structures can be stored in arrays by using references

Operators

- All the standard binary operators from C
  `+` - addition `-` - subtraction `/` - division `*` - multiplication
- Additional operators
  `**` - exponentiation `.` - string concatenation
  `x` - string repetition `..` - range operator (`1..3`⇒`1,2,3`)
- All operators have a corresponding assignment operator
  For example `$string .= $add` concatenates `$string` and `$add` and puts the result into `$string`
- Increment and decrement operators
  `++` - increments `--` - decrements
  prefix ⇒ before evaluation, postfix ⇒ after evaluation
- `||` is logical or, `&&` is logical and - These differ from the C operators in that they return the last expression evaluated instead of 0 or 1. You can also use the words `or` or `and`.

Comparison Operators

Perl provides two sets of comparison operators, one for numbers and one for strings

It's your responsibility to use the right one (the `-w` flag helps)

Function	Numerical	Character
Equality	`==`	`eq`
Non-equality	`!=`	`ne`
Less than	`<`	`lt`
Greater than	`>`	`gt`
Less than or equal	`<=`	`le`
Greater than or equal	`>=`	`ge`
Comparison	`<=>`	`cmp`

Quoting Operators

While we generally think of quotes as literal values, perl provides operators which provide similar capabilities. These are especially useful if the object to be quoted contains quotes.

- `q{`*`text`*`}` - similar to the single quote ('). No variable interpolation is performed.
- `qq{`*`text`*`}` - similar to the double quote ("). Variable interpolation is performed
- `qx{`*`text`*`}` - similar to the backquote (`). After variable interpolation, the *`text`* is treated as an operating system command, and its result is returned.
- `qw{`*`text`*`}` - returns an array of words from *`text`*, equivalent to quoting each one. No interpolation is performed.

You can use any character you choose in place of the {} pair.

Simple Examples

- Print only lines with exactly five fields

  `perl -ne 'print if split == 5'` *filename*

- Print lines that are longer than 80 characters

  `perl -ne 'print if length > 80'` *filename*

- Print the 10th through 20th lines of a file

  `perl -ne 'print if $. >= 10 && $. <= 20'` *filename*

- Print lines where the third field is 0

  `perl -ne 'print if (split)[2] == 0'` *filename*

- Change `cat` to `dog`

  `perl -pe 's/cat/dog/'` *filename*

- Translate lowercase to uppercase

  `perl -pe 'tr/[a-z]/[A-Z]/;'` *filename*

Perl Functions 1

Note: Most functions operate on `$_` if they have no arguments
For more information, type `perldoc -f` *functionname*.

- `chop` - removes last character of its argument
  Useful for removing newlines:

```
while(<>){
    chop;
     ...   # $_ no longer contains a newline
    }
```

- `split` - break a line into words

```
@T = split;                # breaks $_ into words,
@T = split(",");           # uses comma as separator
@w = split(/[ ,]+/,$in); # uses comma or space
```

Perl Functions 2

- `join` - combine an array or list

```
print join(" ",@T);
$fn = join("/",@parts);
```

- `keys` - produce array containing the indices of an associative array

```
@thekeys = keys(%array);
```

- `values` - produce ordinary array containing the values of an associative array
- `each` - iteratively returns a key/value pair from an associative array, or a null value at the end of the array.

```
while(($key,$value) = each(%array)){ ... }
```

iterates over all the key/value pairs.

Perl Functions 3

- `index` - returns the position of the first occurence of a substring within a string. (returns -1 if no match)

```
$str = "the cat in the hat";
$i = index($str,'cat');
```

sets `$i` equal to 4.

- `substr` - extracts part of a string from a larger string

```
$piece = substr($str,3,5); # 5 chars, start at pos. 3
$part  = substr($str,3);   # from pos. 3 to the end
$end   = substr($str,-5);  # starts 5 chars from end
```

- `grep` - evaluate an expression for each element of an array, and return an array of elements for which the expression was true

```
@joes = grep(/joe/,@array);
```

Perl Functions 4

- `map` - returns a list resulting from evaluating an expression on each element of a list.

  ```
  @cts = map(length,@text)
  ```

  `@cts` will contain the lengths of each element of `@text`.
- `eof` - returns 1 if the next input (<>) would return end of file.
- `eval` - evaluate a string as if it were a Perl program
- `unpack` - breaks a string into an array using format codes
  Can be used for fixed format records or binary data

  ```
  @T = unpack("x3A4A5",$in); # like (3x,a4,a5)
  @N = unpack("d4",$vals);   # reads 4 doubles (binary)
  ```
- `pack` does the opposite of unpack

Perl Functions 5

- `push` - add an element to the end of an array
  `push(@array,$newval);`
- `pop` - remove and return the last value of an array
  `$lastvalue = pop(@array);`
- `shift` - remove and return the first value of an array
  `$firstvalue = shift(@array);`

Programming in Perl

- All statements must end in a semicolon (`;`)
- Blocks of statements are surrounded by curly braces (`{ }`)
- Control statements from C: `for`, `while`, `goto`
- Additional statements: `unless` and `until`
- Inside loops, use `next` and `last`
- Compound statements can eliminate the need for brackets

Examples:

```
$x = 3 if $a < $b;
$new = &sub($new) until ($new > $old);
```

- `foreach` statement makes processing arrays easy

  Example: `foreach $value (@array){ ... }`

Reading Input

- Basic input operator is angle brackets: `< >`
- Perl is clever about figuring out where input comes from
  - Reads from files if their names appear on the command line
  - Reads from standard input otherwise
- `open` function creates a filehandle

```
open(FI,"filename") || die "Couldn't open filename";
```

Then read from `filename` with

```
$in = <FI>;
```

- Character based input can be performed using the `read` and `seek` functions:

```
read(FILEHANDLE,$target,$length);
$success = seek(FILEHANDLE,$position,$whence);
```

Two Special Cases

- With a `while` statement, input goes to `$_`

```
while(<>){  # or while(<FILEHANDLE>)
    ...     # each line is read into $_
  }
```

  This is so useful, it's implemented as the `-n` flag.

- If a wildcard is in the brackets, it is expanded and `$_` set to each filename

```
while(<*.c>){
    open(FI,$_);
        . . .
    }
```


Filehandles

- Special cases:

```
open(FH,"<filename");  # open for input only
open(FH,">filename");  # open for output only
open(FH,">$filevar");  # same using variable
open(FH,">>filename"); # open for append
open(PI,"command|");   # open pipe from 'command'
open(PI,"|command");   # open pipe to 'command'
```

- Automatic Filehandles: `STDIN`, `STDOUT`, `STDERR`

```
$response = <STDIN>;
chop($response);
```

- Input record separator defined by automatic variable `$/`
- Setting `$/ = 0777` causes the whole file to be read into a scalar
- Reading a filehandle into an array reads the whole file

References

References are a way of storing a “link” to one variable as the value of another variable, and can be used to produce structures of arbitrary complexity.

Perl will never automatically dereference your references – you always need to explicitly dereference them.

To explicitly make a variable a reference, you can use the backslash (\) operator:

```
$thearray = \@array;
```

To dereference a reference, surround the reference name with curly braces (`{ }`), and precede it with the appropriate “operator”:

```
$value = ${$thearray}[2];    # extract a scalar
@vals  = @{$thearray}[2..5]; # extract an array slice
```

You can check to see whether a variable is a reference using the `ref()` function, which returns `""` for a regular variable.

References (cont’d)

One of the most useful ways of using references is to create an array of arrays, similar to C’s multiply subscripted arrays. In these examples, the arrays are created implicitly.

```
@{$test{"harry"}} = (5,3,2,9);    # makes an array
print $test{"harry"}[3];          # prints 9

$x[1][3] = 17;                    # creates x on the fly
```

When an element of an array is a reference to an array, you must dereference it correctly to use it as an array:

```
push(@{$test{"harry"}},17);
```

Double subscripts are shorthand for the “->” operator:

```
print $test{"harry"}[2];      # prints 2
print $test{"harry"}->[2];    # same as above
```

Example: Rearranging output from `rusers`

The UNIX `rusers` command lists users on each machine on a network. We'll rearrange the output to give a list of users and the machines to which they are logged in:

```
#!/usr/local/bin/perl
open(RU,"rusers `stathosts`|");
while(<RU>){
        chop;
        @T=split(' ');
        ($machine = shift(@T)) =~ s/(.*)\..*/$1/;
        foreach $t (@T){
                push(@{$usr{$t}},$machine)
                        if(!grep(/^$machine$/,@{$usr{$t}}));
                }
        }
@users = sort(keys(%usr));
foreach $u (@users){
        printf("%s%s%s\n",$u,length($u) < 8 ? "\t\t" : "\t",
                join(" ",@{$usr{$u}}));
       }
```


`rusers` Example (cont'd)

The `rusers` output looks like:

```
alpha.Berke joe fred sam
beta.Berkel fred fred fred fred harry
gamma.Berke sue john sam
                ...
```

The output from the `perl` script looks like:

```
fred            alpha beta
harry           beta
joe             alpha
john            gamma
sam             alpha gamma
sue             gamma
               ...
```

Example: Finding and killing jobs

Suppose we want a command which will find telnet sessions which we have initiated and selectively eliminate them.

The UNIX `ps` command can list the telnet sessions. We could then present the process id to the `kill` command. We'll use perl to automate the process.

```
#!/usr/local/bin/perl
@ps = `/usr/ucb/ps auxww | grep telnet`;
foreach $ps (@ps){
        @T = split(" ",$ps);
        if($T[10] =~ /^telnet/){
                print("Kill session to $T[11]? ");
                $resp = <STDIN>;
                chop $resp;
                kill 15,$T[1] if($resp eq "y");
                }
        }
```


Printing

Remember that variables are expanded inside of quotes, making many printing tasks simple.

- `print` function
  - `$,` is field separator
  - `$\` is record separator
- `printf` function behaves like its counterpart in C
- You can specify a filehandle as an alternative to `STDOUT`

```
print STDERR "Error in input line $.\n";
```

- Simplified printing of errors and warnings
  `warn` - prints message (with newline) to standard error
  `die` - like warn, but exits after printing

More on Printing

- Use escape characters to literally print special characters

```
$cost = 100;
print "The cost is \$$cost\n";
```

  results in

```
The cost is $100
```

- Include large blocks of text using perl's "here-is" syntax:

```
print <<identifier;
      ...
  text to be printed
      ...
identifier
```

- Extensive formatted printing can be done with `format/write`

Sorting

The `sort` function by default performs a lexicographical sort.
Alternatively, a subroutine or block of statements can be provided.
Plain sort gives lexicographical sort:

```
@snames = sort(@names);
```

Using `<=>` gives numerical sort (note no comma):

```
@svals  = sort({$a <=> $b} @values);
```

Sorting an associative array based on keys:

```
@skeys = sort(keys(%array));
foreach $key (@skeys){
     push(@sarray,$array{$key});
     }
```

Sorting keys based on the values of an associative array:

```
@skeys = sort({$array{$a} <=> $array{$b}} keys(%array));
```

Functions which return Arrays

- You can use a list with individual members instead of an array

```
($first, $last) = split(" ",$name);
```

- You can extract just part of a returned array

```
# first word from $_ is stored in $first
   $first = (split)[0];
# second through fifth words in $line
   @some = (split(" ",$line))[2..5];
```

- Setting a scalar to an array results in the number of elements

```
# $num is number of comma-separated items in $line
   $num = split(",",$line);
```


Examples

- Finding Unique Lines in a File

```
#!/usr/bin/perl -n
# -n puts the perl program in a while(<>) loop
  print if($x{$_}++ == 0);
```

- Finding the Longest Line in a File

```
#!/usr/bin/perl -n
chop;              # remove the newline
$ll = length;
$maxl = $ll if $ll > $maxl;
print "$maxl\n" if eof;
```

Example: Keeping Track of Number of Fields

```
#!/usr/bin/perl
while(<>){
        $n = split;
        $ct{$n}++;
        }

@skey = sort({$a <=> $b} keys(%ct));

foreach $key (@skey){
        print "$key:\t$ct{$key}\n";
        }
```


Regular Expressions

- `=~` operator allows testing or substitution

  `$in =~ m/r_e/        # returns 1 if r_e  is in $in`

  `$in !~ m/r_e/        # returns 1 if r_e is not in $in`

  (The `m` is optional; you could use `/r_e/` instead.)

  `$str =~ s/old/new/;       #  changes old to new in $str`

  `($new = $save) =~ s/old/new/;#   doesn't change$save`
- Variables are expanded inside of regular expressions
- Regular expressions can be arguments to `split` or to set `$/`

Note: With the `m/.../` or `s/.../` operators, you can replace the "/" with any character you choose, or you can use pairs of {} or `[]` as delimiters. (Avoid using `?` as a delimiter).

Constructing Regular Expressions

Regular Expressions can contain any combination of:

- Literal Characters
  Note: Always escape `. ^ $ + ? * ( ) [ ] { } | \`
- Character Classes
  Characters surrounded by square brackets (`[ ]`)
- Negated Character Classes
  First character in brackets is a caret (`^`)
- Escape Characters

`\n` newline	`\f` formfeed	`\t` tab
`\b`* word boundary	`\d` digit	`\w` alphanumeric
`\B` non-boundary	`\D` non-digit	`\W` non-alphanumeric

- Octal or Hexadecimal Characters
  `\`*nnn* - octal   `\x`*nn* - hexadecimal

* - Represents a backspace in a character class

Constructing Regular Expressions(cont'd)

- Operators within Regular Expressions

`^`	anchors expression to beginning of target
`$`	anchors expression to end of target
`.`	matches any single character except newline
`|`	separates alternative patterns
`()`	groups patterns together
`*`	matches 0 or more occurrences of preceding entity
`?`	matches 0 or 1 occurrences of preceding entity
`+`	matches 1 or more occurrences of preceding entity
`{`*n*`}`	matches exactly *n* occurrences of preceding entity
`{`*n*`,}`	matches at least *n* occurrences of preceding entity
`{`*n*`,`*m*`}`	matches between *n* and *m* occurrences

Grouping in Regular Expressions

In addition to grouping alternations, parentheses are used in regular expressions to tag sub-expressions which can be referred to later. The expression contained by the *i*-th set of parentheses (counting from the left) is referred to as `\i` on the left-hand side of a substitution, and `$i` anywhere else. If you want to use parentheses for just grouping, without tagging the enclosed sub-expression, use `(?:` and `)` for grouping.

For example, the following program identifies the first occurrence in each line of input of two identical words in a row:

```
if(/\b(\w+) +(\1)/\b){
    print '$1 appears twice in line $.\n';
    }
```

Note that the tagged expression is referred to as `\1` in the regular expression itself, but as `$1` outside the regular expression.

Modifiers for Regular Expressions

A number of letters can be placed after the final delimiter of a regular expression to modify some aspects of the matching process

- `i` - Makes the match case-insensitive.
- `g` - Operates on all occurences of the regular expression, instead of just the first, which is the default behavior.
- `s` - Allows `.` to match newline along with everything else
- `m` - Changes the meaning of `^` and `$` anchors so they are meaningful for each logical line. (You can *always* use `\A` and `\Z` to mean the beginning and end of the input.)
- `e` - Treats the right hand side of the `s/.../.../` operator as a Perl expression instead of simply text.
- `x` - Ignore whitespace and comments inside of the regular expression - useful for formatting complex regular expressions.

Return Values from Regular Expression Operators

The values returned from the `m/.../` and `s/.../.../` operators are different depending on the context in which they are used

- `m/.../` operator
  - When used as a scalar, returns 1 if the match was successful, 0 otherwise
  - When used as an array, returns a list of tagged items.
- `m/.../g` operator
  When used as an array, returns a list of tagged items corresponding to as many matches as were found.
- `s/.../.../` operator
  In scalar or array context, returns the number of substitutions which actually took place.

Examples of Regular Expressions 1

Change `NA` to `.` (period)

```
s/NA/./
```

Problem: `NAME` gets changed to `.ME`
Solution: Use `\b` to specify word boundaries

```
s/\bNA\b/ . /
```

The `s` command returns a count of the number of substitutions which are carried out, so it can be used to count occurrences of regular expressions:

```
while(<>){
   $i = s/\bNA\b/ . /g;
   print "Too many missing in line $.\n" if $i > 5;
   }
```

Examples of Regular Expressions 2

Reverse the order of two words

```
s/(\w+) (\w+)/$2 $1/;
```

```
$line = "one two three four";
$line =~ s/(\w+) (\w+)/$2 $1/; ⇒ two one three four
$line =~ s/(\w+) (\w+)/$2 $1/g; ⇒ two one four three
```

Find a word followed by a number

```
if(/(\w+) (\d+)/){
    print "word is $1, number is $2\n";
    }
```

Note that `$1` and `$2` retain their values after the regular expression.

Examples of Regular Expressions 3

Add one to a series of line numbers at the beginning of each line.

```
s/^(\d+)/$1 + 1/e;
```

Since we can't add 1 to a number using text, the `e` modifier was used.

Function calls which return appropriate types can also be used with the `e` modifier. For example, to convert numbers to their hexadecimal representation, you could use the `sprintf` function:

```
$str = "125 234 321";
$str =~ s/(\d+)/sprintf("%x",$1)/ge; ⇒ 7d ea 141
```

Greediness of Regular Expressions

Consider extracting URLs from a document. One strategy could be to find strings which start with `http` or `ftp`, followed by a colon (`:`) and which end with a character which is not a valid character for a URL, leading to a pattern like

```
/((?:http|ftp):.*)[^-_\/.~\w]/
```

But if we use this pattern on text such as

```
Go to http://www.info.com and report any problems.
```

we would find that `$1` will be set to

```
http://www.info.com and report any problems.
```

since the end of line allowed a longer match than the address alone.

Greediness of Regular Expressions (cont'd)

One way to solve the problem is to replace the occurence of "`.`", which represents any character, with a more specific character class. In this case, we could use any character which is a valid part of a URL:

```
/((?:http|ftp):[-_\/.~\w]*)[^-_\/.~\w]/
```

A more general approach is to use a question mark (`?`) after the quantifier to tell perl to find the *smallest* string that matches instead of the longest, which is the default:

```
/((?:http|ftp):.*?)[^-_\/.~\w]/
```

Either of these approaches will extract only the URL and not the text which follows.

Iterative Use of the `m/.../g` operator

When used inside a `while` loop, the `m/.../g` operator can be used to process multiple occurences of regular expressions. For example, suppose we wanted to process a file containing email addresses, in order to count how many address were in each domain. (Recall email addresses are in the form *`username@domain.name`*.) The following program would process each address, even if there were multiple addresses on the line:

```
while(<>){
        chop;
        while(/\b(\w+)@([^@ ]+)/g){
                $count{$2}++;
                }
        }
foreach $key (keys(%count)){
        print "$count{$key} addresses from $key\n",
        }
```


More Examples of Regular Expressions

- Remove trailing newline (alternative to `chop`)
  `s/\n$//;`
- Eliminate leading 0s in a line of numbers
  `s/\b0+(\d)/$1/g;`
- Removing the leading portion of a pathname
  `($tail = $path) =~ s#.+/(.*)#$1#;`
- Capitalize the first letter of first word in a line
  `s/^([a-z])/\U$1\E/;`
- Remove comments (`#`) from a line
  `$line =~ s/(.*)#.*/$1/;`

Using Perl to Edit Multiple Files

Two flags are especially useful when using perl as a stream editor:

- `-p` sets up input loop and automatically prints `$_`
- `-i` allows inplace editing (an extension ⇒ backup copy)

For example,

```
perl -pe 's/\bNA\b/ . /g' old.dat > new.dat
```

would change all `NA`s to periods in old.dat, and write to new.dat

```
perl -i.bak -pe 's/\bNA\b/ . /g' *.dat
```

would do the same in place for all files ending with `.dat`, creating copies with `.bak` appended to the names.

This command works well with the UNIX `find` command:

```
perl -i.bak -pe 's#INCL=-I..#INCL=-I.. -I../include#' \
      `find . -name Makefile -print`
```


Interacting with the Operating System

- `system` lets you execute an operating system command
- backquotes (` `) or the `qx{}` operator let you capture output into a perl variable
  - retains newlines
  - multiline output can be captured in an array
  - variable expansion takes place inside of backquotes
- Perl provides functions for many operating system calls `chdir`, `kill`, `sleep`, `wait`, `exec`, `fork`, `unlink`, `mkdir`, etc.
- The associative array `%ENV` contains environmental variables `$ENV{"USER"}`, `$ENV{"EDITOR"}`, etc.

File System Operators

As in many shells, perl offers operators for simplified file queries

Op	Tests	Op	Tests	Op	Tests
`-r`	Readable	`-w`	Writable	`-x`	Executable
`-l`	Link	`-e`	Existence	`-f`	Plain File
`-d`	Directory	`-T`	Text	`-B`	Binary

Example:

```
$prefix = "/usr/local/help/";
$filename = $prefix.$topic;
warn("No help file for $topic") unless -f $filename;
```


Example: Executing Commands on a Network

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

@machines = ("winnie","pooh","kestrel","eeyore");

foreach $machine (@machines){
        $job = `rsh $machine ps aux | grep $ARGV[0]`;
        print "Jobs on $machine:\n$job\n" if($job);
        }
```

Formatted Output

- The format statement allows you to define a template for output produced by the `write` function. Substitutions are made as follows:

`@<<<···`	left justify as many columns as there are `<`s
`@>>>···`	right justify as many columns as there are `>`s
`@|||···`	center as many columns as there are `|`s

- Specify the variables to be printed below the format lines, separated by commas
- Final line of the format should have a single period
- You can specify a header to appear on top of each page (`$=` is the page size)

Example of Format Statement

```
format STDOUT_TOP =
Name                          Office   Extension
.
format STDOUT =
@<<<<<<<<<<<<<<<<<<<<<<<<<< @>>>>>>>     @>>>>>>
$name,$office,$ext
.
while(<>){
     ($name,$office,$ext) = split(",");
     write;
     }
```

Example of Format Statement (cont'd)

The previous example would convert

```
Fred Smith,425,x7743
John Jones,372,x4450
Harold Johnston,421,x4622
```

to:

```
Name                          Office   Extension
Fred Smith                       425       x7743
John Jones                       372       x4450
Harold Johnston                  421       x4622
```


Creating Subroutines

- Define a subroutine using
  ```
  sub subroutine_name{
     statements
  }
  ```
- The last expression evaluated is returned, or use `return`
- Arguments to subroutines are passed as one list through `@_` :
  First arg is `$_[0]`, second is `$_[1]`, etc.
- Arguments other than scalars must be passed to subroutines as references
- Recall by default perl variables are global. The `my` function can create local variables within the subroutine.
- Perl subroutines may be recursive

Using Subroutines

- Call with `&sub(args...)`
  Note that parentheses and arguments are optional
- Include subroutines using the `require` statement

```
require 'file.pl';
```

- The `require` statement searches the directories in `@INC`, which will always contain the current directory.
- To read in a subroutine using require, the last line must be

```
1;
```


Example: Prompting for a Password

```
sub getpass{
   my($prompt) = $#_ < 0 ? "Password:" : $_[0];
   system("stty -echo");
   print(STDERR $prompt);
   $pass = <STDIN>;
   system("stty echo");
   chop($pass);
   printf(STDERR "\n");
   return $pass;
}
1;
```

If the subroutine was in a file called `getpass.pl`, use:

```
require 'getpass.pl';
$secret = &getpass;   # or &getpass("Prompt:");
```

Example: Counting Strings in Files

Suppose we wish to write a subroutine which will accept a list of files as its first argument, a regular expression as its second argument, and which will return an array containing the number of times the regular expression occurs in each of the files. Since arguments to perl subroutines are passed as a single list, the first argument must be a reference to a list. In the subroutine on the next slide, the list will be stored in a local array called `files` through the use of the `my` function. To call a subroutine such as this, a reference must be passed as the first argument, for example

```
@myfiles = ("test.1","data","README");
@cts = &countpat(\@myfiles,"duck");
```


Example: Counting Strings in Files (cont'd)

```
sub countpat {
  my(@files) = @{$_[0]};
  my($pat)   = $_[1];
  my(@res) = ();
  foreach $f (@files){
     open(FI,"<$f") || die "Couldn't open $f";
     $ct = 0;
     while(<FI>){
         $ct++ while s/$pat//;
         }
     close(FI);
     push(@res,$ct);
     }
  return(@res);
}
```

Example: Resolving Links (recursion)

On many systems there are long chains of links, so using `ls` to find where a file really is can be confusing. Since perl functions can be called recursively, they provide a simple solution.

The strategy is to write a function which prints a given filename, and, if it is a link, to follow the `ls` convention of using `->` to point to the link. This process is continued until a file which is not a link is found.

Since many links are stored as absolute pathnames, we'll need a function which takes a relative pathname and makes it absolute. That turns out to be the hard part. The main loop and printing program (`&reslink`) are shown on the next slide; the pathname resolution program (`&resdir`) is on the slide after the next.

Example: Resolving Links (cont'd)

```
# main loop
foreach $name (@ARGV){
        &reslink($name);
         printf("\n");
        }
# resolve links using "->" notation
sub reslink{
        printf("%s",$_[0]);
        if(! -e $_[0]){ printf("(does not exist)")
                }
        if(-l $_[0]){
                printf(" -> ");
                &reslink(&resdir($_[0]));
                }
        }
```

Example: Resolving Links (cont'd)

```
sub resdir{
        my $dir = $_[0];
        my $now = readlink($dir);
        if(substr($now,0,1) ne "/"){
# not absolute pathname
                $i = 1;
                @n = split("/",$dir);
# resolve relative links (..)
                if(index($now,"../") > -1){
                        $i++ while $now =~ s#\.\./##;
                        }
                $now = join("/",@n[0..$#n-$i],$now);
                }
        return $now;
        }
```


Command Line Arguments

- Arguments are stored in the array `@ARGV`
  First argument is `$ARGV[0]`, second is `$ARGV[1]`, etc.
- `&Getopts` subroutine
  - Argument to `Getopts` is a string of characters
  - Options with arguments are followed by `:` in the string
  - Variables of the form `$opt_`*`letter`* are set appropriately
  - Options and arguments are removed from `@ARGV`

```
require 'getopts.pl';
&Getopts("an:x");
```

Running

`program -a -n 100`

will set `$opt_a` to 1, `$opt_n` to 100, and `$opt_x` to `""`

Resources

- Manual page (`man perl`) - provides a overview of the other 25 (!) manual pages describing `perl`
- Books: (Note: 2nd edition is perl5, 1st edition is perl4)
  Learning Perl, 2nd Edition by Randal L. Schwartz
  Programming Perl, 2nd Edition by Larry Wall & Randal L. Schwartz
  Perl Cookbook by Tom Christiansen & Nathan Torkington
  all three published by O'Reilly and Associates
- Internet Newsgroup: `comp.lang.perl`
  FAQ from convex.com (pub/perl/info/faq) (anon. ftp)
- Reference Card (part of *Programming Perl*)
  from CPAN archive (/doc/perlref-5.004.1.tar.gz)

Resources(cont'd)

- Source is available from CPAN archives (try `www.perl.org` or check `http://language.perl.com/CPAN` for more information.)
- The official perl website (maintained by Tom Christiansen) is located at `http://www.perl.com` .
- `http://www.eecs.nwu.edu/perl/perl.html` is another good starting point for finding WWW perl resources.