Perl Programming Fast

Presented by:
Fred Mallett frederm@famece.com
FAME Computer Education
250 Beach Blvd
Laguna Vista, TX 78578
956-943-4040

Seminar Outline

Perl Syntax

Common constructions

A look at some programs from the web (or elsewhere)

What is(n't) Perl like?

Perl is like UNIX Shell scripting
- text file read at execution time (Command interpretive language)
- Can be easily read (can be impossible to read)
- Comments from a "#" to end of line
- Command line capabilities

```
#!/bin/ksh
print "hello world"; exit

#!/usr/local/bin/perl
print "hello world \n"; exit;
```

Perl is not like UNIX Shell scripting
- Program is read completely and compiled at execution time
- Programs are written in blocks

```
foreach (<STDIN>) {
      print "$_   is readable\n" if -r;
      }
```

Perl is like awk
- Anything you can do in awk, you can do in Perl
- There is an a2p utility

Perl is not like awk
- syntax has many differences (*NR* is *$.*)
- Perl has many more capabilities:
  file, dir, process, network
- There are no string limits

What is(n't) Perl like?

Perl is like C Coding
- Free-format language, whitespace mostly arbitrary (blanks, tabs, newlines, returns, and formfeeds)
- Nearly all statements must be terminated by a ";"

```
if ($x == $y) {
  print "values are equal\n";
} else {
  for ($i=1;$i<=5;$i+=2) {
    $x *= $i;
    print "values are: $i $x\n";
  }
}
```

Perl is not like C Coding
- No Main routine required (it is implied)
- No compile necessary
- No libraries or include files required
- No platform dependent binaries
- Many "syntactical shortcuts"

```
open(WORDS,"$ENV{'list'}");
if (-M WORDS < 7) {
    while (<WORDS>) {
      chomp;
      print;
} }
```

Perl program invocation methods:

As a perl script (on UNIX boxes):
  Put *#!/usr/local/bin/perl* as the first characters of script
  Make the file executable
  Execute as a command (located via search path):
    filename

As a perl script (on Win32 boxes):
  1) Use associated file types, then execute as a command:
    filename.pl
    If you also set *pathext* to include *.pl*, you could run the file with just *filename*.
  2) You can also convert a perl script to a Win32 batch file
     with the *pl2bat* program included in the Perl for Win32
     release. (*c:> pl2bat filename.pl*)
  These methods do not allow I/O redirection on the command line (*filename < textfile.txt*).

As a perl program file (on any OS with a command line):
  Don't need the #! declaration, or execute rights
  Execute the program this way:
    perl filename.pl

As an interactive program (good for syntax checking):
    perl
    print "Bye cruel world\n";
    <ctrl>d (<ctrl>z on Win32)

As a command:
    perl -e 'print 2+2;'
    perl -i.bak -p -e 's/^\s+//;s/\s+$//;'
    somefile

In all methods, Perl parses entire perl file, if there are no syntax errors, it enters the compilation phase

Scalar data numbers:

integers and floating point (real)
accepts the complete set available to C programmers

examples of integer constants:

```
17
-1234
0154  #an octal number representing
decimal 108
0xfffe # hexadecimal of decimal 65534
```

examples of reals:

```
1.25
6.7e38
-3.5e-13
```

Scalar data strings:

Any 8-bit character (256 different characters)
This means that Perl can work with binary data
There is no arbitrary string length

Non-interpreted strings, no special meanings (except \' and \\)

```
'abc'              q(abc)          #the letters
a,b,c
'world\n'    q=world\n=     #the letters
w,o,r,l,d,\,n
'and
now'        #a,n,d,newline,n,o,w
```

Interpreted strings, special meanings must be prefixed by a "\"

```
$     #expand scalar variable data
@     #expand list variable data
\\          #backslash
\"          #double quote
\n          #newline
\r          #return
\t          #tab
\f          #formfeed
\b          #backspace
\v          #vertical tab
\a          #bell
\e          #escape
\007        #octal characters, max 077
\x1d        #hex characters, max xff
\cD         #any control character (here, control
D)
\l          #lowercase next letter
\L          #lowercase following letters until \E
or end of string
\u          #uppercase next letter
\U    #uppercase following letters until \E or end
of string
\E    #terminate \L or \U  (the case-shift
operators are
```

typically used to alter variable values)

”abc\n” q(abc\n) q=abc\n= quorsu

Operators for Working with Scalar Values

numeric operators

```
  +  -  *  / #normal arithmetic operators
  **         #exponentiation
  %          #modulo (10%3 = 1). Both operands
             #reduced to integers prior to
  performing
             # So, (11.1 % 2.9)  same as  (11 % 2 =
  1)
```

string operators

```
  .     #concatenation.  ”aaa” . ”bbb” =
  ”aaabbb”
  x     #repetition. Uses two operands: string and
  integer
        #  ”abc” x 2 = ”abcabc”
        #second operand is truncated to an integer
        #  ”abc” x 2.5 = ”abcabc”
        #numbers less than 1 result in an empty
  string
```

logical operators for numbers

```
  used to compare two numbers
  same availability as in C
        <   <=   ==   >=   >   !=    <=>
```

logical operators for strings

```
  used to compare two strings (similar to FORTRAN
  operators)
        lt  le   eq   ge   gt  ne  cmp
```

Separate logical operators needed to resolve: is 5 less than 10?

```
     if ( 5 lt 10 )   # if comparing as strings
     if ( 5 < 10 )    # if comparing by numeric
magnitude
```

Comparison as numbers is different than comparison as strings

Note that string and integer logical operators are approximately opposite of those used by the UNIX *test* command.

Operator Precedence and Associativity

Associativity: Operator: (highest to lowest)

```
( )         Grouping
none              ++   --   (auto-increment and
decrement)
right                  ! ~ - (logical not, bitwise
not, numeric negate)
right             **
left              =~ !~ (binding operators)
left                    *  /  %  x
left                    . +  - (subtraction)
left                    <<  >> (bit shift)
none              -r and others (file test operators)
none              the named unary operators
none                    <  <=  >=  >  lt  le  ge  gt
(”not-equal” operators)
none                    ==  !=  <=>  eq  ne  cmp
(”equal” operators)
left                    &  (bitwise and)
left                    |  ^ (bitwise or,  bitwise
exclusive or)
left                    &&  (logical and)
left                    ||  (logical or)
none              .. (range operator, list
constructor)
right             ?: (ternary if/then/else operator)
right             =  +=  *=   etc... (all assignment
operators)
left                    ,   (comma)
none              the ”list” operators
right       not   (logical negate)
left                    and  (logical and)
left                    or  xor  (logical or, logical
xor)
```

Conversions Between Numbers and Strings

Perl attempts to convert operands to the type required
by an operator

Converting strings to numbers:

leading whitespace and trailing non-numeric characters are ignored:

```
10 + ”  123section99”  == 133;
```

Non-numeric strings convert to 0

Converting numbers to strings results in what would be printed:

```
(4+2) x 2.3 == 66;
```

the ”x” operator forced (4+2) to be converted to a string after the addition

Converting real numbers to integers:
The *int()* operator can be used to extract the integer part of any number:

```
print int(7.2);        # prints 7
```

Note that all numbers are stored as double-precision floating point (*atof()* is used to convert strings).

The scalar undef value

Referencing scalar variables that have not been assigned a value is allowed, though it causes a warning.

These variables are considered undefined.
   The value is zero if used as a number
   The value is ”” (empty string) if used as a string

”*undef*” is also returned by some functions

Literal List data

Lists of values are composed of literals and/or variables and/or expressions separated by commas, typically in parentheses.

Each value is called an element

Examples of lists of scalar constants:

```
(1,2,3)# a 3 element list with the values 1, 2, 3
(”aaa”,7)# a 2 element list with the values ”aaa” and 7
()        # a list with no elements
print 2, ’a’, 4, ”\n” ; # Print expects a list, so no ()’s needed
```

The range operator returns element values which increment by one:

```
(1..5)   # a 5 element list:  1,2,3,4,5
(1..3,10)# a 4 element list:   1,2,3,10
(1.2..4) # a 3  element list:   1.2, 2.2, 3.2
(’a’..’z’)  # a 26-element list: the lower-case alphabet
```

Assignments involving list data

```
($a, $b)=(17, 22); #a becomes 17; b becomes 22
($a, $b)=($b, $a); #swaps the values of a and b
($a, $b)=(6,7,8,9,10);
($a, $b)=(6);
```

Rule: Excess values on the right of = are not involved, while excess variables on the left are given the value *undef*.

Context

Expressions are of only two types: scalar and list.

There are two major contexts: scalar and array (scalar is further divided into string, numeric, and don’t care).

Many operations are sensitive to context and evaluate differently. Some operations supply a particular context, treating their operands as scalar or array values.

If specified in an array context, Perl will promote a scalar value to a single-element array.

If specified in a scalar context, Perl will treat an array value in a manner determined by the operation.

Variable notation

Perl has scalar, list array, and associative array variables.

Variables in Perl use a prefix for both assigning and expanding.

The prefix determines the data type to be returned.

The prefix does not determine the variable type.

Variables in Perl use suffixes when addressing elements in a list.

The suffix determines which variable type is being addressed.

Only when there is no suffix does the prefix determine variable type.

Variables can contain scalar data, references, or lists of either.

Variable names or references can be enclosed in braces.

Perl Prefixes: $ scalar data
@ list data
% hash (associative array) data (pairs)

Perl suffixes: [ ] elements by index from a list array
{ } elements by keys from a hash

$x=4	@x=('a','b','c')		%x=('a',1,'b',2)	
$x[2]	@x[1,2]	@x[0..3]	@x[2,1]	@x[1-3]

```
$x{a}       @x{a,b}     %x
$#x         scalar(@x)        print  @x  .  ”\n”
```

Some very common functions (operators)

<FH>

The diamond operator accepts a Filehandle (contined) argument.

Used in a scalar context it reads up till (and including) the next record separator, used in list context it reads and returns all records till EOF.

The default input record separator is the newline charactor.

```
    $string=<STDIN>;
    $string=<FH_I_Opened>;
    $string=<>;          Reads from command
    line filename
    @list=<>;
    ($infile,$outfile)=<STDIN>; #expect two
    lines, EOF
```

If <> is the only thing specified in a *while()* condition (no variable name referenced), the default variable $_ is used.

The *while()* condition is the only significant construct where <> is treated this way.

```
  while (<STDIN>){
    print “$_\n”; #double space output (2 new
  lines)
  }
```

Note that in the Win32 ports of Perl input and output text file translations are automatic:

When reading, \r\n is changed to \n so you treat data as ”UNIX”.

When printing, \n is changed to \r\n, so again, handle as ”UNIX”.

chomp()

Chomp removes a single newline from the end of a scalar string:

```
$str=”hi\n”;
chomp $str;
print $str;    # Prints hi without a
newline
```

Chomp will accept a list, and act on each element:

```
@x = (”abc\n”, 17, ”def\n”, ”ghi”);
$y = chomp(@x);
```

Gives *@x* the values *”abc”, 17, ”def”, ”ghi”,* and *$y* gets *”2”*

List-type Array Operators

push() pop()

The *push()* and *pop()* operators manipulate the highest numbered element of an array

```
push(@x,$y);
```

adds a new element at the top of array *@x*, the value equal to *$y* this accomplishes the same as: *@x = (@x,$y)*

push() can push an array of values onto an array:

```
push(@x,7,8,9);#adds 3 new elements to array @x
```

```
$y = pop(@x);
```

Removes highest element of array *@x* and places it in *$y*

pop() on an empty array will return the value *undef*

shift() unshift()

The *shift()* and *unshift()* operators manipulate the lowest numbered element of an array

```
$y = shift(@x);
```

Removes the lowest element of array *@x* and places it in *$y*. This accomplishes the same as: *($y,@x) = @x*

shift() on an empty array will return the value undef.

```
unshift(@x,$y); # same as: @x = ($y,@x)
```

Adds a new element to the bottom of *@x*.

unshift() can unshift an array of values onto an array:

```
unshift(@x,7,8,9); # Same as @x = (7,8,9,@x)
```

shift and *push* can be used together to perform a circular shift:

```
push(@array,shift(@array));
```

List-type Array Operators

reverse()

The *reverse()* operator returns the elements in the argument list in reverse order

```
@x = (7,8,9);
@y = reverse(@x);   # @y gets values 9,8,7
```

If used in a scalar context it concatenates all elements, then reverses the characters. This can also be used to reverse the characters in a scalar:

```
$y=”cat”;
$y=reverse($y);  # $y is now tac

@y=(”kit”,”cat”);
$x=reverse(@y);  # $x is now tactik
                 # Note that this joined a list
                 # into a scalar, then reversed
the
                 # scalar, it is not often used
```

sort()

The *sort()* operator returns its arguments in alphabetic sorted order

```
@x = (50,10,4,2,30);
@x = sort(@x);
```

Gives *@x* the values *10,2,30,4,50*

This is an ascending ASCII sort.

More complicated sorts can be coded in Perl, and will be described in the section on ”user functions”.

splice()

splice(ARRAY, OFFSET, LENGTH, LIST)

splice(@x, 2, 3, ”new”, ”data”)

Associative arrays

Associative array elements are built by specifying both the key and value for each element. This can be done all at once:

```
%list = ('ted','123-456' , 'mary','453-1682');
```

in which the items in the list are treated as key/value pairs. Note that the entire array is defined this way. This implies that an associative array can be "cleared" with:

```
%list = ();
```

Alternatively, an associative array can be added to, one element (pair) at a time:

```
$list{'sue'} = '541-1234';
$list{'b'} = 'xyz';
$list{'c'} = 17;
```

in which unaddressed elements remain unchanged.

Note the curly brackets that distinguish the type of variable:

```
$x[$y]# an element of a list-type array
$x{$y}# an element of an associative array
```

Single elements can be accessed with:

```
print "$list{'mary'}\n"    # prints 453-1682
```

Single elements can be assigned with:

```
$list{'ted'} = '451-2341'; # change the value
```

Keys can be any scalar value, but are treated as strings

```
$x{"yy"}   # element key is "yy"
$x{123}    # element key is "123"
```

To add two to the value of the array element used above

```
$x{123} += 2;
```

Referencing an element that has not been defined returns the *undef* value

Associative-array operators

keys()

The *keys()* operator produces a list of all the current keys in an array

```
$x{"a"}=5; $x{"b"}=6;# load associative array
@y = keys(%x);        # @y gets ("a", "b")
                      # -or- ("b", "a")
```

Note that the order in which the keys of %x are placed into *@y* is determined by Perl. It is not necessarily related to the order in which *%x* was built

If used in a scalar context, *keys()* returns the number of elements in the array

```
$z = keys(%x)# $z gets the number of elements in %x
```

values()

The *values()* operator produces a list of the values of all the elements in an array, in the same order as the keys returned by *keys()*

```
$x{"a"}=5; $x{"b"}=6;#load associative
array
@v = values(%x);   # @v gets ("5", "6")
                   # -or- ("6", "5")
```

If used in a scalar context, *values()* returns the number of elements in the array:

```
$z=values(%x); # $z gets the number of elements in
%x
```

reverse()

reverse() can be used to invert an associative array (change values to keys):

```
%aa=('a',1,'b',2,'c',1);
%inv=reverse(%aa); # reverses list equivalent of %aa
print (%inv,"\n"); # prints 1a2b or 1c2b
```

Note that %inv lost an element since the value of ”a” and “c” were identical, and are now being used as keys.

Associative-array operators

delete

The *delete* operator removes a key-value pair from an array, and returns the deleted value.

```
$y = delete $x{”a”};
```

Removes the key/value pair of *%x* with key ”a”, and returns value into *$y*

exists

The *exists* operator returns true if the specified key exists, even if the value is undefined.

```
if (exists $list{$name})
{print ”$name:\t$list{$name}\n”;}
```

Associative-array operators

each()

The *each*() operator returns the key-value pair of an element in an array as a two-item list. Each successive call returns another element.

```
%x=('a',1,'b',2);# load associative array
@y=each(%x); # load "first" key/value of %x into @y
print "@y\n";# prints "a 1" -or- "b 2"
@y=each(%x); # loads "second" key/value of %x into @y
```

When there are no more elements to be accessed, *each*() returns an empty list. If *each* is called after it has returned an empty list, it resets itself, and begins working its way through the array again. Calling *keys*() or *values*() will also reset the *each* calling sequence.

```
%x=(a,1,b,2);   # load associative array
@y=each(%x);    # load "first" key/value of %x into
@y
print "@y\n";   # prints "a 1"
$count=keys(%x);# reset each calling sequence
@y=each(%x);    # load key/value of %x into @y
print "@y\n";   # prints same as above print
```

Assigning a new value to the entire array (%x to the left of the assignment) will also reset the *each*() calling sequence.

Elements added to the array after the first *each*() may not be picked up by further *each*() calls.

Pre-existing elements can be changed after the first *each*() call, but you can't be sure what order Perl will pass individual elements from *each*().

Statements and Flow Control

From the Quick Reference guide:

Every statement is an expression, optionally followed by a modifier, and terminated by a semicolon. The semicolon may be omitted if the statement is the final one in a BLOCK.

Execution of expressions can depend on other expressions using one of the modifiers if, unless, while or until, for example:

```
EXPR1 if EXPR2 ;
EXPR1 until EXPR2 ;
```

The logical operators ||, && or ?: also allow conditional execution:

```
EXPR1 || EXPR2 ;
EXPR1 ? EXPR2 : EXPR3 ;
```

Statements can be combined to form a BLOCK when enclosed in {}. Blocks may be used to control flow:

```
if (EXPR) BLOCK [ [ elsif (EXPR) BLOCK ... ] else BLOCK ]
unless (EXPR) BLOCK [ else BLOCK ]
[ LABEL: ] while (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] until (EXPR) BLOCK [ continue BLOCK ]
[ LABEL: ] for (EXPR; EXPR; EXPR) BLOCK
[ LABEL: ] foreach VAR† (LIST) BLOCK
[ LABEL: ] BLOCK [ continue BLOCK ]
```

Statements and Flow Control

Program flow can be controlled with:

goto LABEL
Continue execution at the specified label.
last [LABEL]
Immediately exits the loop in question. Skips continue block.
next [LABEL]
Starts the next iteration of the loop.
redo [LABEL]
Restarts the loop block without evaluating the conditional.

Special forms are:

```
do BLOCK while EXPR ;
do BLOCK until EXPR ;
```

which are guaranteed to perform BLOCK once before testing EXPR, and

```
do BLOCK
```

which effectively turns BLOCK into an expression.

Regular Expressions

Perl’s regular expressions can be invoked as arguments to several functions, here are three commonly used functions:

m/re/ Returns a true/false about the match, or matched tagged expressions in an array context.

s/re/string/ Standard substitution operator, has several interesting modifiers, as does the match operator.

split(/re/, string) Uses the RE to split the string.

The following characters can be used to make up regular expressions:

. matches any single character, but not a newline without ’s’

(...) groups a series of pattern elements to a single atom

^ matches the beginning of the string. In multiline mode ’m’ also matches after any newline character

$ matches the end of the line. In ’m’ also matches before every newline character.

[...] matches any character in class [^ ...] negates the class.

(... | ... | ...) matches any of the alternatives

(?# TEXT) comment

(?: REGEXP) non back-referencing group

(?= REGEXP) Zero width positive look-ahead assertion

(?! REGEXP) Zero width negative look-ahead assertion

(? MODIFIER) Embedded pattern-match option can be one or more of i, m, s, or

x.

Quantified atoms match greedily. Followed with a ? they are non-greedy:

+ matches the preceding pattern element one or more times.

? matches zero or one times.

* matches zero or more times.

{N,M} match N through M times

({N} means exactly N times; {N,} means at least N times,

{,M} means up to M times.

Regular Expressions

many character classes are defined in a shorthand notation, here are a few of them:

```
\d is the same as [0-9]
\D is the same as [^0-9]
\w is the same as [a-zA-Z0-9_]
\W is the same as [^a-zA-Z0-9_]
\s is the same as [ \r\t\n\f]
\S is the same as [^ \r\t\n\f]
```

There are also some short hands for anchors:

```
\A   beginning          of          string
\Z   end              of            string
\G   location of previous match search
```

Back-references:

\1 ... \n refer to matched subexpressions, grouped with ()

With modifier x, whitespace can be used in the patterns for readability purposes.

Other notes

The function pos(string) returns the character location of the first character matched by the previous match operation against that string.

There are also many special variables set by the match operator:

```
$&   the part of the string matched by the
RE
$‘   the part of the string prior to the
match
$’   the part of the string after the match
```

```
${n} The tagged portion of the match ($1  $2
...)
```

Functions

Functions typically accept one or more arguments, and:

- return a value (*int)*
- or perform an operation with lasting effect (*print*)
- or both (*shift, pop)*

To use functions effectively, you should know a functions:

- calling syntax:
  - do commas or spaces separate arguments?
  - how many arguments?
  - type of arguments?
  - are there default arguments? (like *print* and *chomp*)
- return values
  - if any, and for array and scalar calling context

You can get different return values, or results depending on the which context a function is called in:

```
@list   = function(); # called in array context
($var)  = function(); # called in array context
$var    = function(); # called in scalar context
$list[2]= function(); # called in scalar context
$t{$e}  = function(); # called in scalar context
$var    = (function());
$var    = function(function());
```

The perl manpage *perlfunc* documents this type of information for all functions.

Beware that you can call many functions as operators or functions:

```
print 1+2+4;    # prints 7
print (1+2+4);  # prints 7
print (1+2)+4;  # prints 3
```

In the last example, the () immediately after the *print* operator told it to act as a function, and accept what is in the () as the arguments.

String functions

```
split(/re/,string,limit)

join(EXPR,LIST)

length(string)

index(string, subtring, OFFSET)

rindex(string, subtring, OFFSET)

substring(string, OFFSET, LENGTH)

quotmeta(string)

uc() lc()  ucfirst()  lcfirst()

eval()
```

User functions

A user function (subroutine) is written as:

```
sub  func-name {
   Perl statement;
   Perl statement;
}
```

Function names are maintained in their own namespace.

Function names are global, duplicate names in the same program conflict

Functions are defined in the order in which they are encountered. If a function name is used more than once, the last one takes precedence.

The "*-w*" command-line switch warns about redefined functions.

Functions can be used as part of any expression in a Perl program.

Functions are called by preceding their name with an &

```
&test;      # calls the function named test
      # $ --> single, @ --> plural, % -->
association
      # & means do it
```

To write user functions, you should know how to:

- Return values
- Pass arguments

Passing arguments to functions

Arguments can be passed to a function in the calling statement:

```
$x = 99;
@y = (”a”,”b”);
&fun(17, $x, @y);
```

Within the function, the list of arguments is available in the array ”*@_*” . (This follows the normal convention for naming an array)

```
sub fun {         # if called as above:
  print ”@_\n”; # prints ”17 99 a b”
  print $_[0];   # prints ”17”
  print $_[3];   # prints ”b”
}
```

Note that the *$_* in the above example looks like the variable name used for the current input line, but the subscript *[0]* indicates otherwise.

Function arguments are passed “by reference”, implying that the function can change the values of arguments in the calling statement by modifying the @_ array:

```
sub makeint {
  $_[0]=int($_[0]);    # Assign to argument array
}
$x=12.47;
&makeint($x);          # Call makeint with $x as
argument
print $x;         # prints 12, as $x is now set to
12
```

Passing arguments to functions

You can also do ”pass-by-value” by making local copies of arguments that are passed. (covered later)

The array @_ is local to the function:

Sequence of actions performed when a function is called with an argument list:

```
save any existing value of @_
set a new @_ for this function call
process function
at function completion, restore any pre-existing
@_
```

This allows functions that use arguments to be nested without losing their own calling info.

If a function is called with NO argument list, @_ is not set, and the function sees whatever values this array contained prior to the function call!! Call the function with an empty argument list to prevent this:

```
&sub1;   #Inherits @_ if called from within a
subroutine
&sub1(); #Has its own empty @_
```

Variables in functions

Variable names used in a function are shared with the rest of the program

```
sub test {
 print ”$x”;  # same $x value from outside the
sub
 $x=42;       # When sub ends, main $x is now 42
 $y=47;       # $y now exists in main program
}
$x=”original”;
&test;        # prints ”original”
print ”$x,$y”;# prints ”42,47”
```

The ability to access variables from outside the subroutine, and set variables that are known after the subroutine ends can be a good thing, it can also be a very bad thing.

Using local variables, you can prevent ”cluttering” up the variable namespace in the main program, as well as prevent a subroutine from inadvertantly changing variables.

Local Variables in Functions

Any variable may be declared as local to the function code:

```
sub no_int {
 local($x);      # initializes a local version of $x,
            # with value = undef
 $x=17;     # set local $x to 17
}
```

Similar to the handling of the @_ array in a function, Perl :

- saves any existing value of $x
- establishes a local $x
  - (initial value = undef if not otherwise specified)
- processes the function
- at function completion, restore any pre-existing $x

local() may be used to the left of an equals sign to provide initial values. For example:

```
local($x,$y);
($x,$y) = @_;# assign function arguments to $x, $y
```

which could be written as:

```
local($x,$y) = @_;  # This is pass-by-value
```

As an example of local variables, and recursion, the following function will compute a factorial:

```
sub fact {
  #print "Passed a \$n variable with a value of $n\n";
  local($n) = @_;
  #print "Have a local variable with a value of $n\n";
  ($n<2) ? 1 : ($n * &fact($n-1));
```

```
}
$result = &fact(4);
```

Function return values

Functions return a value equal to the last expression evaluated in the function at run-time.

```
sub ab {
  $a*$b;
}
```

returns the product of $a and $b. This can be called with:

```
$c = &ab;     # $c gets the product of $a and $b
$d = 3 * &ab;    # $d gets the product of $a, $b, and 3
```

Flow of control in the function will determine the last expression evaluated:

```
sub testif {
  if ($x == 1) {
    5;
  } else {
    10;
 } }
```

If *$x* is equal to *1*, the function returns *5*, otherwise *10*.

Return values can also be arrays:

```
sub array {
  (7,8,9+2);
}
@x = &array;     # @x becomes three elements with values 7, 8, 11
$x = &array;     # $x becomes 11 (the last expression evaluated)
($x)=&array;     # $x becomes 7 (the first expression evaluated)
$x=(&array)[1];# $x becomes 8 (the 2nd expression evaluated)
```

Function calling context

The *wantarray()* operator can be used to determine calling context in a function.

This is useful if you want to be able to write a subroutine that can return different values when called in different ways:

```
  @ret=&sub1;# Want an array returned
  $ret=&sub1;# Want a scalar back
```

Here is an example of a subroutine that breaks words based on ":", and returns different things in array verses scalar context:

```
  #!/usr/local/bin/perl
  $line="Grapes:1.47:100:green"; #some sample data

  $cost=&brkstr($line,1);    # called in scalar, 2
  arguments
  print "$cost\n";           # prints "1.47"

  @item=&brkstr($line);      # called in array, 1
  argument
  print "@item\n";           # prints "Grapes
  1.47 100 green"


  sub brkstr {
    # Usage of brkstr:
  brkstr(string_to_split[,desired_word])
    local(@list);
    local($string,$word)=@_;# make local copies of
  arguments
    @list=split(/:/,$string); # split argument 0
  based on ":"

    if (wantarray()){
      @list;        # all "words" if called in
  array context
    }else{
```

```
    $list[$word]; # desired word if in scalar
context
} }
```

Basic I/O

At EOF, <STDIN> returns undef value, so looping through each input line can be done with:

```
while ($x = <STDIN>) {# will loop until EOF on
STDIN
     actions;
}
```

Standard input can be redirected to a script on the command line (but not on WIN32 perl scripts, unless invoked with *perl perl.script*):

```
Perl.script <input.file          #Reads from
input.file
cat file1 file2 | Perl.script    #Reads output of
cat
Perl.script                  #Reads keyboard input
```

For multiple files the command will get more complicated

As a shortcut for reading from input file(s), Perl provides the <> operator:

```
while ($x = <>) {
  actions;
}
```

If the above program is invoked as:

```
Perl.script infile1 infile2 infile3
```

the *while* loop will read through all input files in the order specified.

If no files are specified on the command line, <> will read from standard input

Opening and Using Files

A ”filehandle” names an I/O connection
  STDIN is a filehandle that accesses UNIX standard input
  STDOUT is a filehandle that accesses standard output
  STDERR is a filehandle that accesses standard error

You can also open your own filehandles

Filehandles are maintained in their own namespace

It is recommended that you use uppercase letters and digits to form filehandles

User files may be for reading with:

```
open (FILE1, ”</a/b/infilename”); # or
open(FH,’<infile’);
```

To open a file for writing, use:

```
open (FILE1, ”>outfilename”);
```

To open a file in append mode, use:

```
open (FILE1, ”>>outfilename”);
```

To use output filehandles, use:

```
print FILE1 ”output strings”; # or write or printf
```

All opens return true if successful, false otherwise.

A file that fails to open for reading will return EOF on the first read.

A file that fails to open for writing will **silently** discard any data sent to it.

Therefore a useful construction is:

```
open(FILE1,”/a/b/file”)||die ”can’t open file”;
```

Files can be closed using:

```
close(FILE1);
```

Opening and Using Files

Note that the pathname argument to *open* is shown in double quotes. This makes it an interpreted string, so backslash escapes, and variables are expanded. To prevent this, use single quotes. This is especially important on WIN32 systems. For example:

```
open (FILE1, "c:\a\b\infilename");
```

Would not work, as the *\b* would become a backspace character. Both methods shown below would work fine:

```
open (FILE1, "c:\\a\\b\\infilename"); # Escape the \s
open (FILE1, 'c:\a\b\infilename');    # Use single quotes
```

Note that all pathnames used in the rest of the course need to be adjusted as above for WIN32, this handout is UNIX-centric.

Distinct from currently opened files, there is a "currently selected" filehandle (default value is *STDOUT*). This value is global across an entire program. It specifies where output goes if no filehandle is specified in an output statement. It can be changed with:

```
$oldvalue = select(NEW_FILEHANDLE);
```

where *$oldvalue* gets the filehandle (as a string) in use before the select.

Note that in Win32 systems, if the file you are reading/writing is not a text file, you need to turn off the *\r\n* to/from *\n* translations:

```
binmode(FILEHANDLE);
```

System functions

There are many operators that return file information (there is also the stat function):

```
-r -w -x -e -d -f -z -o -u -s -M -A -C
```

System Functions

Many UNIX OS functions have been made available directly through perl functions. They behave much like the UNIX system calls and commands by the same name, many work on non-UNIX:

chdir(”directory”)

The current working directory can be changed during a Perl run

The effect of chdir goes away upon program completion

opendir(DIRHANDLE,”directory”) &
readdir(DIRHANDLE)

The *opendir*() operator (optionally built into Perl) can be used to open a directory handle so that *readdir*() can read the entire contents of a directory

rename(list)

The *rename*() operator is similar to the UNIX *”mv”* command:

```
rename(”a”,”newdir/a”);# new filename must be specified
```

chmod(mode,list)

The *chmod*() operator performs what *”chmod”* does

chown(UID,GID,list)

The *chown*() operator performs what *”chown”* and *”chgrp”* do

utime(ATIME,MTIME,list)

The *utime*() operator performs some of what *”touch”* does, setting file access and modification times

link(file,linkname) symlink() readlink()

The *link*() operator performs what *”ln”* does

The *symlink*() and *readlink*() operators act on soft links (*ln -s*)

unlink(list)

The *unlink*() operator performs similar to *”rm”*

mkdir(”directory”,mode)

The *mkdir*() operator performs similar to *”mkdir”*

rmdir()

The *rmdir*() operator performs what *”rmdir”* does

Filename Globbing

Filename expansion (”globbing”) is available using angle brackets:

```
@x = </etc/alia*>; # expands as /bin/sh would do it
```

Or, less ambiguously, using the *glob* function:

```
@x = glob( ”/etc/alia*” ); # Note that ”  ” required
```

Used in an array context, the complete list of matching pathnames is returned.

Note that the special characters available and their meaning are not the same as for regular expressions.

In a scalar context, the *glob* construct returns the next item in the list:

```
while ($next = glob(”~/c*”) {
   program statements using $next...
}
```

The above will go though the loop for each value in the filename list, with *$next* set to the successive values.

What is returned from a filename expansion is what the UNIX command ”echo” would return (directories are not expanded)

Filenames beginning with a ”.” are not returned unless specified in the request (even for root)

Multiple patterns are permitted in angle brackets:

```
</etc/bin/* /usr/bin/X11/*>
```

To do this with *glob*, put multiple wildcards in a single string:

```
glob(’/etc/bin/* /usr/bin/X11/*’)
```

Variables are interpolated before filename expansion:

```
<$x/rpc*> # $x is replaced by its value before
expansion
```

Managing arbitrary child processes

Perl provides multiple ways to launch child processes:

system()

‘ ‘ or qx()

via Perl filehandles

fork()

Summary of process launching

Launch Method	STDIN of Process	STDOUT of Process	STDERR of Process	Wait for complete?
system()	Perl’s	Perl’s	Perl’s	Yes
‘ ‘ qx()	Perl’s	returned as string	Perl’s	Yes
open(FH,’c|’) *(for reading)*	Perl’s	connected to filehandle	Perl’s	only by close()
open(FH,’|c’) *(for writing)*	connected to filehandle	Perl’s	Perl’s	only by close()
fork,exec,	User	User	User	User

wait	selected	selected	selected	selected

Some Terms

Package

A package is a named *"namespace"* (Symbol table). You can have an associative array (hash) called *"mystuff"* in every package. Managed with the *package* declaration. You could access the scalar variable *xs* in the package *Mookie*, with *$Mookie::xs*

Library

A set of routines grouped by purpose. Can be stored in a **.pl* file, and brought into a program with the *require* directive. Mostly superceded by modules (though many have been changed to modules, some still exist).

Module

A reusable package defined in a library file that conforms to some rules (suggestions?), named *<package-name>.pm*, that allows the file to be "included" in a program with the *use* directive.

Pragma

A Module that changes the behavior of the compiler.

Reference

A variable can contain "data", or, it can be a reference, which means it contains the address where something is stored in memory. To resolve data from a variable that contains a reference, we "dereference" it. This is very similar to a pointer (used in many languages), except that you cannot perform "math" on the memory address (thus references are considered "safe" pointers).

Packages

By default, all identifier names in a perl program are managed by perl using a single symbol table. Perl refers to such a symbol table with a ”package” name, the default package name being ”*main*”.

You can access names in a package by prefixing *<package>::* to it:

```
$z=”hi”; print $z;   # prints hi
print $main::z;      # prints hi
```

You can create alternative symbol tables as desired, and switch symbol tables via the *package* command.

Names and associated variables known in each package are kept independently from the same name in any other package:

```
$x=7;
package test;
$x=9;
print ”$x\n”;       # prints 9
package main;
print ”$x\n”;       # prints 7
```

The symbol table chosen via a *package* command persists until the end of the block it is part of, but the symbol table in use may be switched as often as desired:

```
$x=7; $y=8;
{ package test;
  $x=9; $y=10;
}
print ”$x $y\n”;     # out of package ”test”
prints 7 8
package test;    # back to the ”test” symbol
table
```

```
print ”$x $y\n”;      # prints 9 10
package main;    # out of ”test”
print ”$test::x $test::y\n  $main::x”;
```

Any name can be qualified to indicate the package where that name is to be found (arrays, hashes, subroutines).

References

References are similar in functionality to the pointers of C.

The syntax for declaring a reference is to precede the target name with a \ (this is like C's &).

For example:

```
$x = \$y;   # $x is now a reference to $y
```

References can be to any Perl string, variable, function or other reference. For example:

```
$ref1 = \"abc";
$ref2 = \@array; #
$ref = \&fun;
```

Once a reference has been created, it may be used as any other variable name:

```
$$ref1   # is a scalar
@$ref2   # is a list-type array
```

Anything that represents a reference may be used this way:

```
${$array[2]}
# @array is an array of references to scalars
```

Note that perl will complain, and terminate the program if you try to dereference a reference (like those described above) as anything other than what it is a reference to.

Trying to do pointer "arithmetic":

```
$x = $ref + 2;
```

will run, but doesn't do anything useful, as *$x* ceases to be a reference. Even saying:

```
$x = $x - 2;
```

will not make *$x* act like $*ref.*

References (continued)

References can be created in flexible ways:

```
($a,$b) = \($x,$y);# two references
```

Some examples of usage:

```
@y=qw(d e f g h);      # construct array y
$ref=\@y;         # $ref refers to @y
print @$ref;      # prints defgh
print $$ref[1];   # prints e
print $#$ref;     # prints 4
$$ref2{$name};    # dereference value from a
                  # reference to an associative
array
```

Getting at individual elements of an array pointed to by a reference can be accomplished another way. The second method below is most commonly used (especially in Perl documentation):

```
$$ref[2]          # prints 'f' (from above code)
$ref->[2]         # so does this
```

This method doesn't work for array slices. For slices:

```
@$ref[0..2]
```

References provide a solution for passing multiple arrays to a function and having the function know the size of each array passed:

```
sub fun {
local($a,$b)=@_;
print "$#$a  $#$b \n";      # prints "2 4"
print @$a;                  # prints "abc"
}
@x=qw(a b c);@y=qw(d e f g h);

&fun( \@x, \@y);            # call fun with a two-
```

```
element
                    # list (of two references)
```

References (continued)

Don't confuse these two operators: -> =>

The first is the infix dereference operator.

The second is a synonym for a comma (called the 'corresponds to' operator):

```
%a=('a',1,'b',2);
```

Could be written:

```
%b=('a'=>1,'b'=>2);
```

Type Globbing

Perl provides a way to allow names to be aliased to other names:

```
@x=qw(a b c);  $x='hi mom'; # Set some x names
*y=*x;                      # make ALL x names y
names
print $y, @y;               # print 'hi momabc'
```

This is especially useful for passing names to subroutines:

```
@x=(1,2,3);
&chngar(*x);  # pass a pointer to a list of names
print "@x\n"; # prints "99 2 3 7 8 9"
sub chngar {
  local(*subnm)=@_;  # assign from list of names
in @_
  print $#subnm,"\n";# prints 2
  $subnm[0]=99;      # changes the first element
of @x
  push(@subnm,7,8,9);
}
```

Note that the *local()* call had to be smart about how to make the connection with what is stored in @_. In fact, it is even smarter than it appears above.

Type Globbing (continued)

The **x* construct refers to all of the above names, and is referred to as a ”type glob” of the name *x*. We can use this as follows:

```
  $x=”abc”;
  @x=(1,2,3);
  &chngs(*x);          # passes *all* ”x” names
  print ”$x\n@x\n”;
  sub chngs {
    local(*subnm) = @_;#look how smart local() must
  be
    $subnm=”def”;      # $subnm aliased to $x
    $subnm[0]=99;      # @subnm aliases to @x
  }
```

or even

```
  @x=(1,2,3);
  local(*y)=’x’; # alias ”y” names to ”x” names
  print ”@y\n”;  # prints ”1 2 3”
```

Multi-Dimensional Arrays

All of the more complex data structures in perl programs (and perl modules) are based on lists of lists in one form or another.

This is often abbreviated as a LOL. See the *perllol* man page or perldoc document for more details than addressed here.

More correctly, a LOL is a list of references to other lists. For example:

```
@a=qw(a b c d e f);
@b=qw(1 2 3 4 5 6);
@lol=(\@a,\@b);
print  $lol[0][2];          # prints c
print  $lol[0]->[5];  # prints f
```

In the example above, the simple lists (*@a* and *@b*) were named arrays. Often we want to create lists, but not using names. These are called anonymous lists, and often a reference to them is created:

```
$list=['a','b','c'];
print $list->[2];     # prints c, so would
$$list[2]
```

Note that we used [] not ().

LOL's can be created similarly:

```
@lol = ([ "bill", "farmer", 19234, 0],
        [ "joe", "bum", 78459, 1]
        ); # note the comma between anonymous
lists
print $lol[0][2]; # or $lol[0]->[2]
```

Or we can create a reference to an anonymous LOL:

```
$lol = [[ "bill", "farmer", 19234, 0],
        [ "joe", "bum", 78459, 1]
        ]; # note the comma between anonymous
lists
```

in which case we access elements like this:

```
print $$lol[0][1];          # prints farmer
print $lol->[1]->[2]; # prints 78459
print $lol->[1][2];         # also prints 78459
```

```
#print $lol[1][2]; wrong!
#print $lol[1]->[2]; wrong!
```

Multi-Dimensional Arrays

Perl allows you to mix list-type and associative arrays in a multi-dimensional notation:

```
%hash=(”ted”,[’demop’,1265,78],”bill”,[’renfie’
,5685,27]);
print $hash{”bill”}->[1]; #prints 5685
print $hash{”bill”}[1];   # same thing
```

The above is a hash of lists. You could also have lists of hashes, hashes of hashes, lists of hashes of lists, etc....

In a ”real” program, values are often assigned through loop statements.

When using modules, you must often create data structures to pass to methods in the module, or you might be given references, or LOL’s containing data the module has developed.

Perl Modules

A Perl Module is a reusable software component.

- As such, they are written according to a set of guidelines that make sure they don't pollute the programs they are designed to aid in developing.
- Many Modules are written "Object" like, and define classes, and methods for accessing the data/operations in the class. They can "export" thingies to your program. You can also "reach into" the module to use pieces of it, or select which pieces you want.
- They are Perl code (But might *"CODE"* in some C)

Perl modules are files named with a *.pm* extension.

They should be placed in Perl's include path, or you can modify the include path, here is how one Win32 version of Perl was setup:

```
$ perl -e '$,="\n"; print @INC'
C:\APPS\perl53\lib\i386-win32
C:\APPS\perl53\lib
c:\fred\perl\lib
.
```

There are many modules "shipped" with perl, called the standard libraries, and, of course, there are the CPAN and other modules.

Win32 ports typically include a set of Win32 related modules.

Modules might add routines, override routines, or overload routines. One even allows you to assign constants, like PI.

It is also possible to *autoload* only the functions you need from a module.

What modules?

There is a list of standard modules in most perl documentation, for example, Appendix B of the Learning Perl books. You could also list the *lib* directory to see what is there:

```
AnyDBM_File.pm  Fcntl.pm        Symbol.pm       constant.pm     newgetopt.pl
AutoLoader.pm   File            Sys             ctime.pl        open2.pl
AutoSplit.pm    FileCache.pm    Term            diagnostics.pm  open3.pl
Benchmark.pm    FileHandle.pm   Test            dotsh.pl        ops.pm
Bundle          FindBin.pm      Text            dumpvar.pl      overload.pm
CGI             Getopt          Tie             exceptions.pl   perl5db.pl
CGI.pm          I18N            Time            fastcwd.pl      perllocal.pod
CORE            IO              UNIVERSAL.pm    find.pl         pwd.pl
CPAN            IO.pm           User            finddepth.pl    shellwords.pl
CPAN.pm         IPC             abbrev.pl       flush.pl        sigtrap.pm
Carp.pm         Math            allmod          ftp.pl          site
Class           Net             assert.pl       getcwd.pl       stat.pl
Config.pm       Opcode.pm       auto            getopt.pl       strict.pm
Cwd.pm          Pod             autouse.pm      getopts.pl      subs.pm
Devel           SDBM_File.pm    bigfloat.pl     hostname.pl     syslog.pl
DirHandle.pm    Safe.pm         bigint.pl       importenv.pl    tainted.pl
DynaLoader.pm   Search          bigrat.pl       integer.pm      termcap.pl
English.pm      SelectSaver.pm  blib.pm         less.pm         timelocal.pl
Env.pm          SelfLoader.pm   cacheout.pl     lib.pm          validate.pl
Exporter.pm     Shell.pm        chat2.pl        locale.pm       vars.pm
ExtUtils        Socket.pm       complete.pl     look.pl
```

This might include modules installed locally, as well as the standard Perl modules.

To add modules from CPAN, load the module (or the tree structure) into a location listed in *@INC* (or add a new location to *@INC*).

Using modules

Modules can provide variables (or various data structures) for use in your program, they might also provide functions you can call.

Modules might provide extra capabilities, or provide ”better” functionality. Some methods can override ”built-in” functions.

You must tell the compiler to ”include” the module:

```
use <Module-name>; # Bring in all exports
use Module ();   # Take no exports, let me call
what I
                 #    want with
”Module::something”
use Module LIST; # Bring in only LIST
```

Depending on how the module was written, and included, the module might ”export” stuff, or you might have to call them with package naming:

```
use Cwd;              # bring in all exported
stuff
print $ENV{’PWD’};    # prints nothing on early
Win32
print getcwd();       # works ok, getcwd is
exported
print Cwd::getcwd();  # Ok also, told which
package
```

An example of overriding (on UNIX systems):

```
print $ENV{PWD};              # Prints /user/fred
chdir (”/”);  print $ENV{PWD}; # Prints
/user/fred
use Cwd(’chdir’);  # Call in chdir, cause not
exported
chdir (”/”); print $ENV{PWD};  # prints /
```

Using modules

Note that some modules are stored in directories under the lib directory. These are called structured packages. For example:

```
C:>cd c:\APPS\perl53\lib
C:>dir Text
Abbrev.pm    ParseWords.pm  Soundex.pm  Tabs.pm
Wrap.pm
```

To use these:

```
use Text::Wrap;
$Text::Wrap::columns=15;
$long=”this is the text to be wrapped into a
format\n”;
print wrap(’’,”\t”,$long); #
wrap(initind,ind,string)
```

Which prints:

```
this is the
        text
        to be
        wrappe
        d into
        a
        format
```

Another example:

```
use Config;
print $Config{’osname’};  # prints MSWIN32
```

Using Objects

Many modules in Perl are written using Object Oriented techniques.

Perls objects are created through creative use of packages, references, and associative arrays (hashes).

An object is a data structure that contains variables you can either dereference or assign, and functions (methods) that you can call.

What we care about here, is that many module methods return a reference to an object. We need to be able to access the properties (methods and variables) of that object. It is done using the -> operator.

The infix dereference operator is used in two ways. The first we have seen when using multi-dimensional arrays:

$ref->[3] $ref->{'$x} $ref->{$u}->{$x}

The second is when accessing properties of an object. Here is an example of accessing a method within an object:

```
$obj=mod1::mthd3($arg1,$arg2);    # module returns an object
print $obj->getinfo("total");    # call the getinfo method
                                 # of the returned object
```

When a module is OO, sometimes you directly access methods within the module, which returns a reference to an object:

```
use CGI;
$query=CGI->new();     # method returns ref to an object
$query->param('date',scalar(localtime)); # set var in obj
```

Some code blurbs

```
  $count=$sum=0;
    while ($x = <STDIN>) {
      chomp($x);
      if ($x eq ”” ) {
          if ($count > 0 ){
              print ”$count non-blank lines in that block\n”;
              $sum+=$count; $count=0;
         }
        } elsif ($x eq ”END”) {
           $sum+=$count;  last;
        } else {
           $count++;
    }  }
    print ”total non-blank lines= $sum\n”;

while (<STDIN>){
      if (/[ \t]$/){
        print ”line $. ends with white\n”; # let
perl count lines
    } }

$count=0;
    while (<STDIN>) {
      $count+= s/\bin\b/$&/g;
    }
    print ”$count\n”;

$/=”\n\n”;
while (<STDIN>){
 write STDOUT;
}
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
$_
.
```

Some code blurbs

```
#!/usr/local/bin/perl
while (<STDIN>){
s{ <!(.*?)(--.*?--\s*)+(.*?)>}{if ($1 || $3)
{"<!$1 $3>";}}gesx;
s{ <(?:[^>'"] *|".*?"|'.*?') +>}{}gsx;
s{ (&( \x23\d+|\w+);?)} {$entity{$2}||$1}gex;
print;
}
BEGIN {%entity = (
        lt     => '<',      #a less-than
        gt     => '>',      #a greater-than
        amp    => '&',      #a nampersand
        quot   => '"',       #a (verticle) double-
quote
        nbsp   => chr 160, #no-break space
        iexcl  => chr 161, #inverted exclamation
mark
        cent   => chr 162, #cent sign
.
.
.
        chr 203, #capital E, dieresis or umlaut
mark
        Igrave => chr 204, #capital I, grave
accent
 .
.
.
       thorn  => chr 254, #small thorn, Icelandic
        yuml   => chr 255, #small y, dieresis or
umlaut mark
    );

    for $chr ( 0 .. 255 ) {
        $entity{ '#' . $chr } = chr $chr;
```

```
        }
}
```

Recursive Subroutine to display a directory tree:

```
   $x=shift;
   chdir($x) || die ”can\’t chdir to $x”;
   ($dev,$ino,$mode,$nlink) = stat(’.’);
   &dodir(’.’,$nlink);
   sub dodir {       # process a directory.
     local($dir,$nlink) = @_;
     local($dev,$ino,$mode,$subcount);
     # Get the list of files in the current
directory.
     opendir(DIR,’.’) || die ”Can’t open $dir”;
     local(@filenames) = sort readdir(DIR);
     closedir(DIR);
     if ($nlink == 2) {#  this dir has no
subdirectories.
       for (@filenames) {
         next if $_ eq ’.’;
         next if $_ eq ’..’;
         print ”$dir/$_\n”;
     } } else {   # this dir has subdirectories.
       $subcount = $nlink - 2;
       for (@filenames) {
         next if $_ eq ’.’;
         next if $_ eq ’..’;
         $name = ”$dir/$_”;
         print $name,”\n”;
         next if $subcount == 0; # Seen all the
subdirs?
         ($dev,$ino,$mode,$nlink) = stat($_);
         next unless -d _;
         chdir $_ || die ”Can’t cd to $name”;
         &dodir($name,$nlink);
         chdir ’..’;
         --$subcount;
```

```
} } }
```