Enhance Your Skillset with Perl

Get tasks done more efficiently and effectively by learning to program with one of the most useful programming languages available: Perl

Copyright Notice

* IMPORTANT COPYRIGHT & LEGAL NOTICE:
*
*

Course Goals

* Learn Perl
* Gain skills needed to use Perl in a practical manner
* Practice using your new skills

What You Get in this Course

* Lectures using PowerPoint slides
* Live examples on Command Line
* Exercises
* Downloadable example code and solutions to exercises

How to Get the Most out of This Course

* Get a notebook
* Take notes during lectures
* Do the exercises
* If possible, try to use what you learn in real life

Course Prerequisites

* Familiarity with files and directories
* Use of the Command Line Interface
* Programming background helps

* On Unix, knowledge of:
  * file permissions and how to change them
  * The PATH environment variable

Section 4
Covers

* Scalars
* Literals
* Numbers
* Strings
* Number and string operators
* If control structure
* While loop
* Etc.

Helloworld.pl

```
#!/usr/bin/perl

print “Hello, world!\n”;
```

Literals

* Values typed into source code

* print “Hello, world\n”; # “Hello, world” is a string literal
* print 7; # “7” is a number literal

Scalar Data in Perl

* Scalar – one piece of data in Perl
* Can be acted on with operators
* Can be stored
* Can be read from a file or written to a file

Numeric Scalar Examples

* Integers
  * 100
  * -5293
  * 93e20 (93 times 10 to the 20th power)
  * 7_393_486_023

Numeric Scalar Examples, CONT'D

* Floating Point
  * 1.073
  * 3.14
  * -8.31E-10 (negative 8.31 times 10 to the negative 10^{th} power)

Numeric Scalar Examples, CONT'D

* Non-Decimal
  * 0144 # 144 octal, which is 100 decimal
  * 0x64 # 64 hex, i.e. 100 decimal
  * 0b1100100 # 1100100 in binary, which is 100 decimal

Numeric Scalar Operators

* + # Addition
* - # Subtraction
* * # Multiplication
* / # Division
* ** # Exponentiation
* % # Modulus (remainder)
  * 10 % 3 = 1

String Scalars

* Sequence of characters
  * Empty string
  * Longest string
  * Any characters

String scalars

* String – a sequence of zero or more characters between quotes

* “”
* ‘abc’
* ‘This is a great day!’
* “Whatever.”
* ‘0123456789’

String Operators

* . # String concatenator
  * “o”.”u”.”8”.”1”.”2”# same as “ou812”

* x # String repetition operator
  * “go“ x 7 # same as “gogogogogogogo“

Single-quoted String Literals

* Strings in single quotes are interpreted literally
  * Except for backslash and another single quote
* ‘stuff’ # Single-quoted string literal
* ‘’ # Null string
* ‘end\n’ # The characters e, n, d, \, and n
* ‘he\’s’ # The characters h, e, ‘, and s
* ‘c:\\Temp’ # The characters c, :, \, T, e, m, p

Double-quoted String Literals

* Strings in Double quotes allow backslash control characters and variable interpolation
* “Hello, world!\n”
* “\n” # newline
* “\t” # tab

Backslash Escape Sequences

\n	Newline
\r	Return
\t	Tab
\f	Formfeed
\b	Backspace
\a	Bell
\e	Escape (ASCII escape character)
\007	Any octal ASCII value (here, 007 = bell)
\x7f	Any two-digit, hex ASCII value (here, 7f = delete)
\x{2744}	Any hex Unicode point (here, U+2744 = snowflake)
\N{CHARACTER NAME}	Any Unicode code point, by name
\cC	A "control" character (here, Ctrl-C)
\\	Backslash
\"	Double quote
\l	Lowercase next letter
\L	Lowercase all following letters until \E
\u	Uppercase next letter
\U	Uppercase all following letters until \E
\Q	Quote nonword characters by adding a backslash until \E
\E	End \L, \U, or \Q

Double-quoted String Literals

* Backslashes in Double quotes will escape
  * $cash # dollar signs, e.g. the beginning of a variable
  * “ # double quotes

* “\”Hello, world!\”\n”; # “Hello, world!”
* “\$cash\n”; # $cash

Auto Conversion between numbers and strings

* Conversion depends on operator used
  * 7 * 7 # equals 49
  * 7x7 # equals “7777777”
  * fred * 7 # equals 0 (‘fred’ would convert to 0)

Scalar Variables

* Holds a single value
* Begin with a dollar sign (called a “sigil”) -- $ -- then a letter or underscore, optionally followed by other letters, digits, or underscores

* $value
* $_ou812_
* $family_member

Getting User Input

* <STDIN> # Takes keyboard input up to a newline

* print “What is your name?\n”;
* $name = <STDIN>;
* print “Hello, $name”;

Chomp

* chomp # Removes a newline from end of input

```
print “What is your name?\n”;
$name = <STDIN>;
print “Hello, $name”;

chomp $name;
print “Hello, $name\n”;
```

Scalar Variables, CONT'D

* Are assigned values with an = sign

* $dessert = "chocolate cake";
* $grade = 97;
* $animal = 'dinosaur';

Operations with variables

* $price = 198;
* $price = $price + 174;
* $avg_price = $price / 2;

* Shortcut…
* $price += 5; # Same as $price = $price + 5;

Data – Binary Operators

+ # Addition, e.g. 2 + 2 = 4

- # Subtraction, e.g. 8 – 4 = 4

/ # Division, e.g. 8 / 2 = 4

* # Multiplication, e.g. 2 * 2 = 4

% # Modulus (remainder), e.g. 10 % 6 = 4

. # String concatenator, e.g. “Sha”.”zam!” = “Shazam!”

x # String repetition, e.g. “go”x7 = “gogogogogogogo”

Binary Assignment Operators

* $price = 2; # $price now equals 2
* $price += 3; # $price now equals 5

* $price -= 1; # $price now equals 4
* $price *= 2; # $price now equals 8
* $price /= 2; # $price now equals 4
* $price **=2; # $price now equals 16
* $price %= 3; # $price now equals 1
* $price .=“ too” # $price now equals “1 too”

Classic Computer Science Problem: Exchange the value of two variables

- Take two user inputs, each separated by a newline.
- Store the first input into variable $a and the second into $b.
- Print out the variables $a and $b.
- Exchange the values of $a and $b using only the variables $a and $b.
- Print out the values of $a and $b again.

The “if” control structure

```
if (comparison expression is true) {
        execute this expression;
}
```

For example:

```
if (‘happiness’ gt ‘depression’) {
        print “happy!\n”;
}
```

The “if” control structure, CONT’D

‘if’ statements can have alternative an option with ‘else’

For example:

```
if (‘happiness’ gt ‘depression’) {
        print “happy!\n”;
} else {
        print “:-(\n”;
}
```

The “if” control structure, CONT’D

‘if’ statements can have multiple options with ‘elsif’

```
if (‘happiness’ gt ‘depression’) {
        print “happy!\n”;
} elsif (‘happiness’ gt ‘sadness’) {
        print “:-(\n”;
} else {
        print “Dang… \n”;
}
```

String Comparison Operators

* Comparison operators return a *true* or *false*
* lt Less than
* le Less than or equal to
* gt Greater than
* ge Greater than or equal to
* eq Equal to
* ne Not equal to

Numerical Comparison Operators

* Comparison operators return a *true* or *false*
* < Less than
* <= Less than or equal to
* > Greater than
* >= Greater than or equal to
* == Equal to
* != Not equal to

Perl “Boolean” Rules for Scalars

* A scalar value is true if it is non-zero; only a 0 is false
* A string is true unless it’s the empty string, i.e. ‘’ or ‘0’

```
If ($value) {
        print “$value\n”;
}
```

Operator Precedence and Associativity

* Precedence – which operators are evaluated first in a complex expression
  * Does 4+3*7 equal 49 or 25?
  * Perl follows common math rules, so answer to above is 25

* Associativity – when operators in an expression have same precedence, do you evaluate left-to-right or from right-to-left?
  * Does 4**3**2 equal 4096 or 262144? 262144 (4 to the 9^{th} power)

* UNIX - See the perlops page by typing "man perlop"
* Windows – Start->All Programs->Strawberry Perl->Related Websites->Perl Documentation
  * Then in the left-hand column, under "Reference", click on the "Operators" link

Use Parentheses to Create Your Own Precedence and Associativity

* By default: 4+3*7 equals 25
* But, (4+3)*7 equals 49

* By default: 4**3**2 equals 262144 (4 to the 9th power)
* But, (4**3)**2 equals 4096

* Parentheses make order of evaluation clear

Exercise # 1

* 1. Write a program that asks the user to input two numbers. Then the program takes the two numbers and outputs the result of the following binary operators on the two numbers:

  +

  -

  /

  *

  %

  .

  x

Exercise # 2

* 2. Write a program that asks the user to input two numbers, then prints out which number is the greater of the two. If the two numbers are equal, either can be printed out.

Exercise # 3

* 3. Write a program that asks the user to input one number.
* Next print out the sum of the number added to itself (in other words, double the number).
* Then multiply the sum you just printed out by itself (in other words, take the sum and raise it to the 2^{nd} power).
* Finally, concatenate the number to itself.

CHALLENGE Exercise!

BONUS Exercise
Create a Program that calculates daily caloric needs
1st – Basal Metabolic Rate (BMR)

The Harris-Benedict formula (BMR based on total body weight)

* Men: BMR = 66 + (13.7 X wt in kg) + (5 X ht in cm) - (6.8 X age in years)

* Women: BMR=655+(9.6Xwtinkg)+(1.8Xhtincm)-(4.7Xageinyears)

* Note:
  * 1 inch = 2.54 centimeters
  * 1 kilogram = 2.2 lbs.

* Example:
  You are male
  You are 35 yrs old
  You are 5' 9 " tall (175.3 cm)
  You weigh 180 lbs. (81.8 kilos)
  Your BMR = 66 + 1121 + 876.5 - 238 = **1825.5 calories/day**

BONUS Exercise, Continued

2nd

Use the BMR, along with the activity factors below, to calculate calorie expenditures for a day

* Activity factor:

Sedentary = BMR x 1.2 (little or no exercise, desk job)
Lightly active = BMR x 1.375 (light exercise/sports 1-3 days/wk)
Mod. Active = BMR x 1.55 (moderate exercise/sport 3-5 days/wk)
Very active = BMR x 1.725 (hard exercise/sports 6-7 days/wk)
Extr. Active = BMR x 1.9 (hard daily exercis/sports & physical job or 2 X day training, marathon, football camp, contest, etc.)

undef

* undef # the value of a scalar variable before it is given a value

* A variable that is undef will act like a 0 if used as a number

defined

* The defined function will return true if a variable is defined

```
print “What is your name?\n”;
chomp ($name = <STDIN>);
if ( defined($name) ) {
        print “Hello, $name\n”;
}
```

Lists and Arrays

* List – an ordered collection of scalars
* Array – a variable that contains a list

* Can contain any number of scalars
* Indexed from first (index 0) to last item in the list

List Literals

* List literal – list of comma-separated values in parentheses

```
(1, 2, 3)                          # list of three integers

("Huckle", "berry", "Finn")  # list of strings
qw( Huckle berry Finn )        # same as above using qw shortcut

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)       # list of ten integers
  (1..10)                                 # same as above using .. (the range operator)
```

List Assignment

```
# Assignment to variables
($mom, $dad, $son) = (“Bill”, “Sarah”, “Johnny”);

# Assignment to an array
@family = qw( Bill Sarah Jonny);
```

Array Operators

- push and pop – Push values onto, and pop values off of the end (highest index value) of the array

```
@family = qw( Bill Sarah );
push @family, “Laura”;        # @family now has Bill, Sarah, and Laura.

$youngest = pop @family;      # $youngest eq “Laura”
```

Array Operators, CONT'D

shift and unshift – Push values onto, and pop values off of the beginning (lowest index value) of the array

```
@family = qw( Bill Sarah Johnny);
$oldest = shift @family;        # $oldest eq "Bill"

unshift @family, "Bill";  unshift @family, "Grandma";
$oldest = shift @family;        # $oldest eq "Grandma"
```

Array Operators, CONT'D
splice

splice – allows addition or deletion of values in the middle of an array.

* Takes up to 4 arguments – 2 mandatory; 2 optional
* 1st argument is the array splice is operating on
* 2nd argument is the position in the array to start. If no other arguments are given, splice will remove and return everything from its starting point to the end of the array.

```
@numbers = (1, 2, 5, 7, 9);
@others = splice @numbers, 2;       # @others = qw(5 7 9);
```

splice, CONT'D

splice – a 3rd argument will specify a length

```
@numbers = (1, 2, 5, 7, 9);
@others = splice @numbers, 2, 2;    # @others = qw(5 7);
```

splice – a 4th argument can be used to provide a replacement list

```
@numbers = (1, 2, 5, 7, 9);
splice @numbers, 2, 3, 3..9;  # @numbers = qw (1 2 3 4 5 6 7 8 9);
```

Arrays – Accessing Values

```
@family = qw( Bill Sarah Jonny);
```

- With indices

```
$family[0] == “Bill”;                    # true
$family[1] == “Sarah”;                   # true
```

- With special indices

```
$#family                                 # last element index, i.e. 2
$family[$#family] == “Johnny”            # true
```

Arrays – Interpolation Applies

```
@family = qw( Bill Sarah Jonny);

print “$family[0] is the oldest family member\n”;  # Will print
        # Bill is the oldest family member

print “The whole family is: @family\n”;
        # The whole family is: Bill Sarah Jonny

print “\@family = @family\n”;
        # @family = Bill Sarah Jonny
```

Arrays – Operators

```
@family = qw( Bill Sarah Jonny);
@numbers = (0..9);

$sum = $numbers[5] + $numbers[9];
print “The sum is $sum\n”;    # The sum is 14

$fourBills = $family[0]x4;       # BillBillBillBill
```

Arrays – Operators, CONT'd

@numbers = (1..5);

- The reverse Operator

@reverseNumbers = reverse(@numbers); # gets 5, 4, 3, 2, 1

@numbers = (2, 1, 5, 4, 3);

- The sort Operator

@numbers = sort @numbers; # gets 1, 2, 3, 4, 5

The foreach Control Structure

* foreach loops through a list of values

```
foreach $number (1, 2, 3, 4, 5) {
        print “$number\n”;
}

@numbers = (1..5);
foreach $number (@numbers) {
        print “$number\n”;
}
```

The each Operator

* **each** returns an index and its corresponding value

```
@numbers = (0..9);

while (($index, $num) = each @numbers) {
        print "$index  $num\n";
}
```

The while control structure

* The while control structure creates a loop that repeats while a condition is true

```
$value = 10;
while ($value) {
        print “The value is $value\n”;
        $value -= 2;
}
```

while Exercise

* Create a program that prints out powers of 2, from 2 to the 0^{th} power up to 2 to the 10^{th} power. The exponentiation operator is "**". So, 2 raised to the 5^{th} power would be: 2**5. The output of your program should look like this:

* 2 to the 0 power is 1
* 2 to the 1 power is 2
* 2 to the 2 power is 4
* 2 to the 3 power is 8
* 2 to the 4 power is 16
* 2 to the 5 power is 32
* 2 to the 6 power is 64
* 2 to the 7 power is 128
* 2 to the 8 power is 256
* 2 to the 9 power is 512
* 2 to the 10 power is 1024

Scalar and List Context

* As Perl processes each expression in your program, it always expects the statement to result in either a scalar or a list value. Perl's expectation is the "context" of the expression.

* 77 * $value; # Scalar context
* foreach $value (@array) { } # List context

Forcing scalar context

* The scalar operator forces Perl to interpret an expression in a scalar context.

```
@qbs = qw(Brady Manning Palmer Newton);
print “There are “, @qbs, “quarterbacks playing today.\n”;

print “There are “, scalar @qbs, “quarterbacks playing today.\n”;
```

<STDIN> in List Context

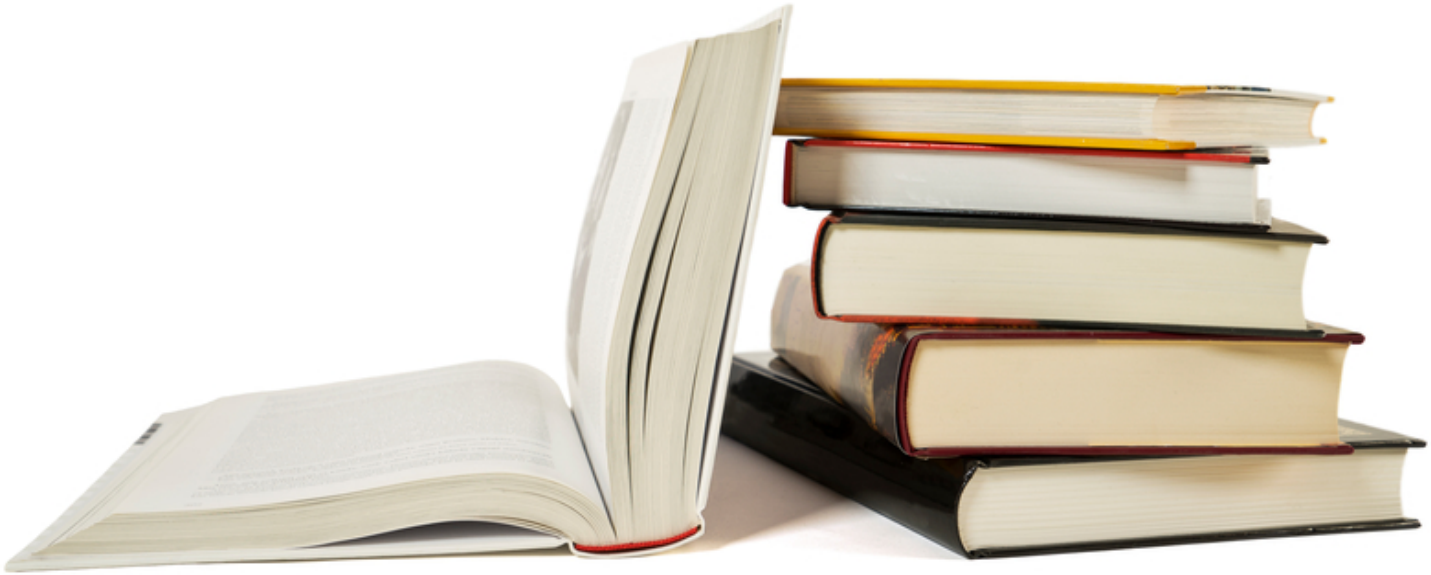
* @inputLines = <STDIN>;

In list context, <STDIN> returns all lines, one line at a time, until the end of file or input (ctrl-D in Unix; ctrl-Z in Windows)

chomp @inputLines; # Chomps all newlines in the array.

Exercises

* 1. Create an array that contains a list of numbers from 1 to 25. Then, in a single print statement, print out how many elements the array contains, followed by all of the numbers in the array on a separate line with a space between each number and a newline after the last number.
* 2. Reverse the array you created for exercise # 1. Print it out just as you printed out the array in exercise # 1.
* 3. Print the even numbers from your array on one line. Print the odd numbers on a separate line.
* 4. Copy the contents from your array into a second array. Reverse the second array. Next, for each index, multiply the value of the two arrays together and print out multiplied value from the first to the last index.



Subroutine Example compoundInterest.pl

* Program contains two subroutines
  * &getInfo
  * &compoundInterest

* Subroutines organize and/or shorten programs

Subroutines

* Subroutines are user defined functions.
* Subroutine name can start with letters and/or underscores.
* Subroutine name can contain letters, digits, underscores.
* Subroutine name sometimes has a ‘&’ in front of it.
* Can be placed anywhere in your program.

Invoking a Subroutine

- Invocation is referred to as “calling” the subroutine
- Return value is the value of the last expression evaluated in the subroutine

```
&getInfo;       # subroutine called in a “void context”
```

Subroutine Arguments

```
# Subroutines can have arguments passed to them
&compoundInterest ($principal, $intRate, $num, $years);

# Arguments are held in the @_ array in the subroutine
# E.g. in &compoundInterest, the @_ array contains:
$_[0] == $principal;
$_[1] == $intRate;
$_[2] == $num;
$_[3] == $years;
```

Variable Length Argument Lists

- You could determine length of argument list with this statement:

```
$argLength = @_;        # @_ array in a scalar context
```

- Or you could use the special indice "$#_";

```
$numEntries = $#_ + 1;
print "The number of entries in the \@_ array is $numEntries\n";
```

- Or go through arg list with a foreach loop

```
foreach $arg (@_) {
        statements;
}
```

Private Variables

* The “my” operator make variable private to any enclosing block
* Enclosing blocks of subroutines, if, while, foreach structures, etc.

```
sub compoundInterest {
        my $P = shift;
        my $r = shift;
        my $r = $r/100;
        my $n = shift;
        my $t = shift;

        $P*(1 + $r/$n)**($n*$t);
* }
```

The “use strict” Pragma

* Pragma = a hint to the compiler
* The “use strict” pragma tells the compiler to enforce good, i.e. “strict”, programming rules.
* It’s recommended to use it

The return operator

* “return” returns immediately from a subroutine

```
sub positiveValue {
        if ($value > 0) {
                return $value;
        }
        -1;
}
```

Return values

* Can be scalar or list values

```
($principal, $intRate, $num, $years) = &getInfo;      # returns a list

$value = &compoundInterest;                           # returns a scalar
```

Persistent, Private variables

* The “state” operator

```
sub addCrew {
        state $n =0;
        state @crew;

        push @crew, @_;
        $n += @_;
        print “$n crew members are present:\n\t@crew”;
}
```

“state” variables restriction

* Can’t initialize state arrays and hashes in list contexts as of Perl 5.10

Exercise
The Compound Interest program

* The formula for annual compound interest is **A = P (1 + r/n) ^ nt:**
* **Where:**
* **A** = the future value of the investment/loan, including interest
  P = the principal investment amount (the initial deposit or loan amount)
  r = the annual interest rate (decimal)
  n = the number of times that interest is compounded per year
  t = the number of years the money is invested or borrowed for

* Assignment: write a program that collects P, r, n, and t from a user, then calculates what the ending value would be. The program should have two subroutines – one that collects information from the user; the other that does the calculation

Getting info into and out of your computer

* Let's start with input

Input

* Getting info into your Perl program:

```
<STDIN>                    # The line input operator

$line = <STDIN>;           # Returns next line of input in a scalar context

foreach ($line = <STDIN>) {          # Returns all lines of file as a list
  print “The line now is: $line”;
}
```

while loop shortcut

```
while ($line = <STDIN>) {                    # Returns all lines of file as a list
   print “The line now is: $line”;
}
#==============================#
while (<STDIN>) {
          print $_;
}
# Uses the $_ variable, Perl’s favorite default variable
```

The Diamond Operator

* <> reads contents of files listed as invocation arguments. Returns undef at the end of input.

```
# cat.pl
while (<>) {
        print;
}

cat.pl morpheus neo trinity      # Prints out morpheus neo trinity file contents
```

Invocation Arguments

* Contained in the @ARGV array
* @ARGV can be accessed and manipulated from within your Perl program
* @ARGV is the list the diamond (<>) operator searches for command line arguments
* Before using <> in your program, you can set @ARGV if you wish.

```
@ARGV = qw (morpheus neo trinity);
```

Output
print

* Getting info out of your Perl program:

* print

* Prints a list of items as strings to Standard Output

```
print “There are “, 268, “ days until the 2016 election.\n”;
print “@array”;                    # Interpolation
print @array;                      # No interpolation
```

print CONT'D

```
print("hot"."inHere"."\n");  # with parenthesis is a function call

print (3 + 4);
```

Formatted Output
printf

* printf – like print, but takes formatting strings called “conversions”.
* Conversion strings begin with a percent sign, i.e. “%”
* Number between % and conversion string causes left or right justification.

```
$color = “red”;
$number = 7;
printf “Wow!  You have %5g pairs of %s shoes!\n”,  $number, $color;
```

%s – strings
%g – guesses what right (floating point, interger, etc.) for numbers
%d – decimal numbers
%f – floating point numbers

Input and Output with Filehandles

* A filehandle is a connection between your Perl program and the outside world.
* Named like variables and arrays, i.e. begins with a letter or underscore, which is optionally followed by additional letters, underscores, and/or digit
* 6 special filehandles: STDIN, STDOUT, STDERR, ARGV, DATA, ARGVOUT

* STDIN – keyboard by default
* STDOUT – computer monitor by default
* STDERR – computer monitor by default

```
program.pl < inputFile 2> outputFile;   # manually set STDIN, STDOUT, STDERR
```

Input and Output File Descriptors

* A file descriptor is an integer that a Unix-based operating system uses to identify files being accessed by a program.

* STDIN – 0. Can be redirected with "<".
* STDOUT – 1. Can be redirected with ">".
* STDERR – 2.

```
program.pl < inputFile  >outputFile  2> errFile;
# manually set STDIN, STDOUT, STDERR
```

Filehandles, CONT'D

- Open filehandles with the 'open' function

```
open INPUT, 'file';                    # open file for reading
open INPUT, '<file';                   # open file for reading
open OUTPUT, '>$outputFile';  # open file for writing using variable
open TRASH, ">>", "$bitbucket";        # open file for appending
```

Filehandles, CONT'D

* Close filehandles with 'close', by letting program end, or by reopening.

* Bad filehandles return end-of-file (undef in scalar context and empty list in list context).

* Use –w or warnings pragma to get help from Perl to detect bad filehandles.

die, warn, and autodie

* Use the ‘die’ function to quit Perl program, print an (optional) error, and return a non-zero exit status (exit status is 0 when program executes successfully).

```
if (! open FILE, ‘>>’, ‘$message’) {
        die “Can’t open $message for appending: $!”;
}
```

* Use the ‘warn’ function to deliver a message and line number where the error occurred, like ‘die’ does, but program continues.

* Perl 5.10 and later contain the ‘autodie’ function

```
use autodie;
open FILE, ‘<‘, ‘whateverFile’;
```

Filehandles, CONT'D

* Once they're open, use filehandles just like you used <STDIN>

```
if (! open FILE, '>>', '$output') {
        die "File $output couldn't be opened for appending: $!";
}
print FILE "Whoa!  You're cookin' with gas!\n";
```

Changing the Default output location

* Use the ‘select’ operator to choose a default output location other than STDOUT

```
open FILE, “>outputFile”;
select FILE;
print “Hello, world!\n”;        # output goes to FILE
select STDOUT;                  # sets output filehandle back
```

Changing Default STDERR location

* Change STDERR location by reopening STDERR

```
if (! open STDERR, “>> /var/tmp/errlog”) {
        die “Couldn’t open /var/tmp/errlog for appending: $!”;
}
```

Printing output with ‘say’

* The ‘say’ operator is just like the ‘print’ operator except that it appends a newline.

```
print “Hello, world\n”;
say “Hello, world”;

# both of the above statements cause identical output.
```

Using scalar variables for filehandles

* Using a scalar variable for a filehandle allows:
  * Passing filehandles as arguments to functions
  * Storing filehandles in arrays or hashes

* Create a filehandle in a scalar by using a scalar variable as the argument to 'open'. For example:

```
open $file_fh, ">>", "./outputFile";
print $file_fh "Luke, I am your father!\n";
```

Exercise # 1

1. Modify compound interest program so that when it prints out the future value of the investment or loan, it does so with two digits after the decimal, and a $ sign before the amount.
2. Additionally, make the compound interest program print out the difference between the principal amount and the future value amount. Do this using one additional print statement (i.e. avoid using any more variables).

Exercise # 2

* Write a program that takes two command line arguments. The first is the name of a file to take as input. The second is optional, but if it's given, it's the name of the file you want to output the data from input file to. If the second argument isn't given, then output the first file's data to Standard Out.

Hash

* Hash – a data structure that can hold multiple values, where those values are indexed by strings rather than numbers.
* The keys are related to the values.
* Indices are called “keys”.
* The keys are arbitrary, i.e. they can be ANY string.

Hash example

```
%ticker_to_co;

$ticker_to_co{aapl} = “Apple Inc.”;
$ticker_to_co{orcl} = “Oracle Corporation”;
$ticker_to_co{fb} = “Facebook”;
$ticker_to_co{dis} = “The Walt Disney Company”;
$ticker_to_co{amzn} = “Amazon.com Inc.”;
```

Hash and Array Differences

* Percent sign % preceding name refers to whole hash, whereas a whole array uses ‘at’ symbol @.

%ticker_to_coName;

* Hashes use curly braces { } instead of square brackets.

$ticker_to_coName {aapl} = ‘Apple Inc.’;

* Values are NOT stored in order.

Hash Similarities to Arrays

* Hash names – start with underscore or letter, followed by optional underscores, letters, or numbers.

* Keys are unique

* Hash can be any size

* Keys can be expressions

$ticker_to_coName {‘aa’.’pl’} == $ticker_to_coName {aapl}

Hash Assignment

* List of key/value pairs in list context

```
%ticker_to_coName = (aapl, “Apple Inc.”, fb, “Facebook”);
```

* “Big Arrow” a.k.a. the “Fat Comma” is easier on the eyes

```
%ticker_to_coName= (   aapl => “Apple Inc.”,
                       fb => “Facebook”,
                       dis => “The Disney Company”,
                   );
```

Hash Assignment, CONT'D

* "Unwind" hash with assignment to array.

```
@tickers_and_cos = %ticker_to_coName;
```

* Reverse hash assignment is possible

```
%coName_to_ticker = reverse %ticker_to_coName;
```

Hash Functions

* **keys** function lists all of a hash's keys.
* **values** function lists all of a hash's values.
* **keys** and **values** functions list in same order.
* In scalar context, **keys** and **values** return number of key/value pairs.

Hash functions, CONT'D

* The **each** function iteratively returns all key/value pairs in a hash.

* **each** is typically used in a loop like this:

```
while ((my $ticker, my $company) = each %ticker_to_co) {
        print "$ticker\t$company\n";
}
```

Hash Functions, cont'd

* The **delete** function deletes a key and the corresponding value

```
delete $ticker_to_co{aapl};
```

Hash Interpolation

* Hash elements are interpolated when placed between double quotes.

```
print “aapl ~ $ticker_to_co{aapl}\n”;          # prints “aapl = Apple Inc.”
```

* Unlike whole arrays, whole hashes do not interpolate

```
print “%ticker_to_co\n”;                      # prints “%ticker_to_co”
```

Hash Exercise # 1

* You're a teacher. Here are your students and their grades. Store the names and grades in a hash. Use **sort** to print out the list of students from the highest to lowest grade.

Elmer Fudd
73
John Winkle
82
Heinz Doofinshmirtz
62
Patrick Star
27
Pearl Krabs
85
Wile E. Coyote
100
Melissa Duck
93

Excercise

* Read in contents of the file named, “mcDs1perLine”. Get the values in a hash so that the keys are the meal name and the value for each meal is its price. Lastly, print out each meal and its price on one line.

Hash Exercises # 2, 3

* Read in the contents of the Huckleberry Finn file. Create a hash that contains a count of each word in the file. Print out how many times the word “and” is used in the file. Print out how many times the word “table” is used in the file. Print out how many times the word “brother” is used in the file.

Intro to Regular Expressions

* Regular Expression – a group of letters and/or symbols that allow you to match patterns in a string.

* Regular Expressions have a single purpose – to match text.

* They either match or they don't.

What Text Do You Match and How?

* Text in $_

* Match by placing the regex, or ***pattern***, between forward slashes:

```
$_ = “The word regex is short for the phrase, ‘regular expression’”;
if (/regex/) {
    print ‘We have a regex!\n’;
}
```

* Variables between slashes are interpolated

```
$word = “hello”;
/$word/             # is equivalent to /hello/
```

Regular Expression examples: Metacharacters

* Examples of regular expressions:

  . – the dot matches any single character except a newline

  * -- the asterisk matches 0 or more occurrences of the preceding character, for example .* matches ANY string.

  + -- the plus symbol matches 1 or more of the preceding character

  ? – the question mark match 0 or 1 occurrences of the preceding character

* NOTE: The backslash -- \ -- erases the powers of metacharacters

Grouping Regular Expressions

* Parentheses () group regex patterns, for example:

/Perl is (so,? *)+ cool!/ -- matches “Perl is so, so, so cool!”

* Parentheses also allow reuse of matched patterns by using “back references”. A back reference is a backslash followed by a number.

/Te(.)\1e(.)\2(.)\3/ -- matches the word “Tennessee”

Back References CONT'D

/(.)\17-93(.)\2/ -- Is the first back reference for '1' or for '17'?

* Perl 5.10 introduced \g{N}, where 'N' is the number of the back reference you want to use.

/(.)\g{1}7-93(.)\g{2}/ -- matches the number "777-9311"

Regex Alternatives (the 'OR' bar)

* The vertical bar – "|" – gives you alternatives. If the pattern to the left of the bar doesn't match, then try the pattern on the right of the bar.

/this and|or that/ # matches "this or that" as well as "this and that"

/Te(.)\1e(s)\2(.)\3|Mi(.)\4i(.)\5i(.)\6i/ --matches Tennessee or Mississippi

The Character Class

* Character Class – a list of possible pattern matches enclosed in square brackets '[ ]'.
* Only on of the characters in the brackets will match.

* /[0123456789]/ – matches a single number.
* /[0-9]/ -- same as the above, using a hyphen '-' to specify a range.

* A carat '^' negates what's between the square brackets.
* /[^0-9]/ -- matches any characters EXCEPT integer numbers.

Character Class Shortcuts

* \d – was same as /[0-9]/, i.e. it matched any single decimal number
  * /\d+/ -- matches any decimal number
* \s –matched any whitespace, i.e. /[\f\t\n\r]/
* \w – matched /[a-zA-Z0-9_]/

* Character Class shortcuts were simple prior to Perl 5.6. At that time,

* Starting with Perl 5.14, append /a to the end of match operator to match ascii, which is pre-5.6 functionality.

/\d+-\d+/a -- matches 777-9311 using only ASCII characters

Negation of Character Class Shortcuts

* \D – matches any non-digit character, i.e. [^0-9]
* \S – matches any non-whitespace character, i.e. /[^\f\t\n\r]/
* \W – matches [^a-zA-z0-9_]

* [\d\D] – matches any character

Exercises

* 1. Write a program that prints any line that contains words with repeating letters.

* 2. Write a program that asks the user for a word to search for in a file, then prints out every line from the file that contains the word the user chose.

Pattern Match Operator

* m// is the pattern match operator
* m/pattern/ is same as /pattern/
* You can use different delimeters with 'm//'
* m/stuff/ is same as m%stuff% is same as m{stuff}
  * m<http://> is much more clear than /http:\/\//

* Interpolation

```
$movie = "Avengers";
m/$movie/        # equivalent to m/Avengers/
```

Match "Modifiers"

* i – makes matching case insensitive

/Perl/i -- matches "Perl" and "perl" and "pErL" and "PERL"

* s – makes . match newlines

/.*/s -- matches "any string\n of characters \n including \n newlines"

* \N – matches everything except \n. So, if you wanted to match newlines with \s up to a certain point, you could use \N to do so:

$_ = "any string\n of characters \n including \n newlines"

/any .*including\N/s # matches up to \n after 'including'

* x – causes Perl to ignore whitespace in a pattern

/Te (.)\1 e (.)\2 (.)\3/x -- allows whitespace to make patterns easier to read

Match “Modifiers” CONT’D

* You can combine match modifiers

```
$_ = “Peter Piper picked \na peck of pickled peppers”;

if (/  peter    .*    peppers  /isx) {
            print “Yep, Peter picked ‘em!\n”;
}
```

Anchors

* \A – anchors pattern to start of string

$_ = ‘Nanny is Nanny and I call her Nan…’;
m{\A(Nanny) is \1} # matches

* \z – anchors pattern to the absolute end of string, i.e. no \n after it.
* \Z – anchors pattern to the end of a string with optional \n after it.

m{\A(nan).*\1 .*\1.*\z}i

Anchors CONT'D

$_ = "Start, then go until the end";

* ^ is same as \A

/^Start/ # matches $_

* $ is same as \Z

/end$/ # matches $_

Anchors CONT'D

/m modifier changes behavior of ^ and $ to a line-by-line basis

$_ = "The key is to\nStart, then go until\nThe end\n is reached";

* With /m, ^ matches at beginning of string or immediately after \n

/^Start/m # matches $_

* With /m, $ matches at end of string or immediately before \n

/end$/m # matches $_

Word Anchors

* \b – word boundary anchor, where a word contains letters, underscores, and/or numbers.

```
$_ = “We grow and flourish”;
/we\b/i          # matches $_

$_ = “Weeds grow and flourish”;
/we\b/i          # doesn’t match $_
```

* \B matches where \b doesn’t

```
$_ = “Weeds grow and flourish
/we\B/i          # matches $_
/we\b/i          # doesn’t match $_
```

Exercise

1. Write a “grep” program that will search a file for a single word that you specify, and that will take a “-i” command line argument which will cause it to ignore the case of the word you search for. When it finds a string with the word you’re looking for it prints that string to standard out (the screen). Two examples of the syntax for the command look like the following:

- ./grep.pl word file
- ./grep.pl –i word file

The Binding Operator

* =~ is the binding operator
* =~ gives you freedom to match strings other than those in $_
* =~ tells Perl to match the pattern on the right of =~ against the string on the left

```
print “Is it hot, cold, or warm where you are?\n”;
chomp ($temp = <STDIN>);
if ($temp =~ /hot/) {
        print “It’s getting $temp in here!\n”;
}
```

Match Variables (aka “Captures”)

* Capture what you match in match variables ($1, $2, $3, $4, etc.)
* For each pair of parentheses, there’s a match variable
* $1 is for first pair of parentheses, $2, for the second, etc.

```
$_ = “Spinach is Popeye’s favorite food.”;
if (/^(\w+)\s+.*\s?(\w+).$/) {
        print “Popeye ate his favorite $2 for dinner -- $1!\n”;
}
```

Match Variables, CONT'D

* Persistence – match variables persist until next *successful* match
* Most matches are used in **if** or **while** loops

* Non-capturing parentheses: (?:)

```
$_ = "My shoes are brown";
if (/My\s+(?:tennis )?shoes.*\s(\w+)/) {
        print "Your shoes are $1\n";
}
```

Automatic Match Variables

* Perl provides three automatic match variables, each with a weird name
* $` -- contains the portion of the string before match
* $& -- contains the portion of the string that matched
* $’ -- contains the portion of the string after the match

```
$_ = "one two three”;
if (/\s\w+\s/) {
      print "Before match:($`)\nMatch: ($&)\nAfter match: ($')\n";
}
```

General Quantifiers

* The * and the . and the ? were all “quantifiers”.
* Other quantities of matches can be specified with:
  {number, number}

```
$_ = “The little girl screamed, \”weeeee\” as she slid down the sliding board\n”;

if (/(we{2,5})/) {
    print “$1 is a common exclamation when utilizing a sliding board”;
}
```

Regular Expression Precedence

* Parenthesis () have highest precedence
* Quantifiers * . ? + {num1, num2} are next
* Anchors \A, \Z, \z, ^, $ come next
* 'Or' | is next in the precedence line
* Individual characters, character classes, and back references, collectively called "atoms" are last in precedence. Examples are: abc, [abc], \1

More info on Perl Regular Expressions

* Study these manpages:
  * perlre
  * perlrequick
  * perlretut

The Substitution Operator

* s/// is the substitution operator
* Matches the regex between the first two forward slashes; replaces what is matched by what is between the last two forward slashes.

* s/regular_expression/replacement/

```
$_ = “Popeye’s favorite food is spinach.”
s/spinach/broccoli/
print;              # Will print “Popeye’s favorite food is broccoli.”
```

s/// examples

* $_ = “Popeye loves\nspinach and Olive Oyl\n.”

```
s/ oyl /oil/ix;                 # Notice the ‘i’ and ‘x’ modifiers.  The ‘s’ modifier works also.
s/olive oil/more spinach/;      # Popeye loves spinach and more spinach.

s/^spinach/Olive Oyl/m;         # Popeye loves Olive Oyl and more spinach.

s/spinach$/Olive Oyl/m;         # Popeye loves Olive Oyl and more Olive Oyl.
s/Olive Oyl/squash/g;           # ‘g’ (global) modifier modifies all matches on a line
                                # $_ now says, “Popeye love squash and more squash.
```

Delimiters and Binding

* You can use delimiters of your choosing

s!pattern!replacement!

s<pattern><replacement> # Use 2 sets when using open & close delimiters

* The binding operator can be used

$veggie = “potato”;

$veggie =~ s/pot/tom/;

Case Shifting

```
$_ = “all caps”;
s/(.*)/\U$1/;      # $_ now is “ALL CAPS”.  \U affects all chars after it

$_ = “LOWER CASE”;              # \L causes lower case
s/(.*)\s+(.*)\n/\L$1 \E$2/I;    # \E ends \U, \L affects

$_ = “mr. mike masters”;
s/(. *\b)/\u$1/g;               # Makes only next char upper case.
s/(. *\b)/\l$1/g;               # Makes only next char lower case
```

The split operator

* split splits a string based on a pattern and returns a list of strings.
* split /pattern/, $string

```
@values = split /,/, “5,4,3,2,1”;     # Returns list of five strings, each a number

@fields = split /:/, “daemon:*:1:1:System Services:/var/root:/usr/bin/false”;

@words = split /\s+/, “We hold these truths to be self-evident”;

split;                  # Splits $_ on white space.
```

The join operator

* join joins a list based with joining character.

join $joiner, @joinee;

@values = split /,/, “5,4,3,2,1”; # Returns list of five strings, each a number

$add = join “+”, @values;

Regular Expression's Greediness

* $_ = "Programmers program in Perl to solve their problems."

```
m/(program).*(\1)/i;                    # greedy match
```

* A ? after a quantifier makes it non-greedy, i.e. ? Makes the quantifer match as few characters as possible for a match

```
m/(programm).*?(\1)/;                   # non-greedy and more efficient match
```

* *?
* +?
* ??
* {2,7}?

Exercise

* 1. Write a program called, “addLines.pl” that reads in the file, “PG_IOPS_abridged.txt”. Then, for each line read, “addLines.pl” will record the line number, discard the date, and add all the numbers on the line together. Print each line out to the screen. Also, print each line out to a file called, “outputFile”.

Exercise

* 2. Write a substitution program called, “sed.pl”. This program will behave like a very limited version of the unix sed command. The syntax will look like this:

    sed.pl ‘s/phrase/newPhrase/’ file
    sed.pl ‘s/phrase/newPhrase/g’ file
    sed.pl ‘s/phrase/newPhrase/gi’ file

* By default, the sed.pl program will replace the first occurrence of “phrase” it finds in “file” with “newPhrase”. If a trailing letter “g” is given, the program will replace ALL occurrences of “phrase” in “file” with “newPhrase”.

The $^I variable

```
$^I = .bak;
while (<>) {
        s/perl/Perl/g;
        print;
}
#####################################
# The $^I variable moves file into ‘file.bak’, then creates ‘file’ and prints results into original file name.
```

unless

* unless executes a block of code if a condition returns false. Opposite of **if**.

```
unless ($full) {
        &eatFood;
}
```

* Is the opposite of the **if** conditional.
* Can use the **else** clause.

```
unless ($cat) {
        &feedDogFood;
} else {
        print “Here kitty, kitty.\n”;
}
```

until

* Opposite of while. Executes block until condition is true.

```
my $cat = 0;
my $dog = 1,000,000,000,000;
until ($cat >= $dog) {
        $cat = $cat + 1;
}
```

Expression Modifiers, i.e. Control Structures after Expressions

```
&eatFood unless $full;

print “Dogs are great!\n” if $dog > $cat;

&buyKitten if $daughterBegs;


my $cat = 0;
my $dog = 10;
$cat += 2 until $cat == $dog;
```

- Only one expression allowed.

Naked Control Block

* Open and close curly braces

```
{
   your code;
   your code;
}
```

* Allows you to limit the scope of **my** variables.

Autoincrement and Autodecrement ++ and --

```
$count = 0;        print "$count";          # prints 0
$count++;          print "$count";          # prints 1

$nextCount = $count++;                     # postincrement
print "$nextCount";                        # prints 1
print "$count";                            # prints 2

$nextCount = ++$count;                     # preincrement
print "$nextCount";                        # prints 3
print "$count";                            # prints 3

$count--;                                  # postdecrement
print "$count";                            # prints 2
```

The for control structure

* Same as in C

```
for ($i = 0; $i < 10; $i++) {
        print "$i\n";
}
```

* The for and foreach keywords are the same in Perl. Perl discerns each by looking at what's inside the parentheses that follow them. If there are two semi-colons, it's a for loop. If there are none, it's a foreach loop.

Loop Controls

* Loop controls end execution of innermost currently running loop. Loops include for, foreach, while, until, or the naked block.
* The loop controls are: last, next, redo

* last – ends loop immediately

```
while (<>) {
        if (/whatever/) {
                s/whatever/anything/;
                last;
        } else {
                &doStuff;
        }
        &doMoreStuff;
}
```

* next – immediately starts next iteration of the loop

* redo – jumps back to the beginning of the loop ***without*** going to the next iteration.

Loop Controls Work on Innermost Loop

```
while (<>) {
    chomp ($num = $_);
    for ($i = 0; $i <= $num; $i++) {
            if ($i == 2) {
                    print “double the fun\n”:
                    next;
            } else {
                    print “$i\n”;
            }
    }
    last if $i == 7;
    }
```

Labeled Blocks

* Labeled blocks allow you to jump from one part of your Perl program to another specified part of your program.
* Labels are just like other Perl identifiers, i.e. they're created using one or more letters, numbers, and/or underscores, but they can't start with a number.
* Labels are rare. I don't remember ever using them.
* Here's an example:

```
TOP: while (<>) {
        foreach (split) {
                last TOP if /$something/;   # End if $something is discovered
        }
    }
```

The ternary operator ?:

* Same as in C
* condition ? True return value : false return value

```
$goat = $brady > $montana ? $brady : $montana;
print "The G.O.A.T. is $goat!\n";
```

* To create "case"-like conditional structure

```
my $activityFactor =
  ($activityLevel == 1) ? 1.2 :
  ($activityLevel == 2) ? 1.375 :
  ($activityLevel == 3) ? 1.55 :
  ($activityLevel == 4) ? 1.725 :
  ($activityLevel == 5) ? 1.9 :
                          1.375;          # default is level 2
```

Logical Operators

* The logical operators are:
* && # the AND operator
* || # the OR operator

```
if ((ur Happy) && (uKnowIt)) {
        &clapUrHands;
}

if ((ur Happy) || (uKnowIt)) {
        &stompUrFeet;
}
```

Partial Evaluation Operator Control Structures

```
my $value = $first || (2+3);      # returns 5 if $first is 0, else it returns $first
```

```
$weight > $goal && &diet;
$weight > $goal and &diet;              # Same as above
($weight > $goal) && (&diet);           # More clear
```

&& can be replaced with ‘and’

|| can be replaced with ‘or’

Installing Modules

* Use one of the following installation methods:

1. CPAN.pm module
2. cpan script
3. CPANPLUS

Module exercise

* Download and import the Spreadsheet::WriteExcel module.

* Create a spreadsheet as is shown in the “Synopsis” portion of the module description.

File Tests

* File tests give you particular information about a file

* A File test is a one-letter operator that is preceded by a hyphen, like this: -A

* Use file tests by placing them in front of a filename in a control structure.

```
if (-e $filename) {
        print "$filename exists!\n";
}
```

File Tests

* There are many different file tests. Here are some of them:

-r	Tests if file or directory is readable
-w	Tests if file or directory is writable
-x	Tests if file or directory is executable
-e	Tests if file or directory exists
-z	Tests if file or directory exists and has zero size
-f	Tests if entry is a plain file
-d	Tests if entry is a directory
-M	Tests modification age (measured in days)
-A	Tests access age (measured in days)
-s	File or directory exists and has a nonzero size (returns size in bytes)

Complete list is in accompanying file

Multiple File Tests on Same File

* How can you perform more than one test on the same file?

```
print “The plain file, $file, exists!” if (-e $file and –f $file);
```

* Or, use Perl’s “virtual filehandle”, _

```
print “The plain file, $file, exists!” if (-e $file and –f _);
```

* Or, as of Perl 5.10, stack multiple file tests

```
use 5.010;
print “The plain file, $file, exists!” if (-e –f –x –o $file);
```

File test Exercise 1

* Create a program similar to the “fileHandle.pl” program. Your program should create a file called “testfile” if a file by that name doesn’t already exist.
* If “testfile” exists, then your program should create testfile1.
* If testfile and testfile1 exists, then your program should create testfile2. If testfile, testfile1, and testfile2 all exist, then your program should create testfile3. In other words, your program should increment an integer until it can create a unique file name by using that integer at the end of the file name.

File test Exercise 2

* Create a program that determines and lists a file's type. The program should do this for each file listed on the command line when your program is invoked. Use the file tests below to determine file type.

-f	File is a plain file
-d	File is a directory
-l	File is a symbolic link
-S	Entry is a socket
-p	Entry is a named pipe
-b	Entry is a block-special file
-c	Entry is a character-special file

The stat and lstat Functions

* stat – Returns complete profile of information about a file

my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize, $blocks) = stat ($file);

==========

$dev – the device number of the file

$ino – the inode number of the file. The combo of $dev and $ino uniquely identifies a file.

$nlink – the number of hard links to the file or directory.

$mode – the set of permission bits for the file.

$uid, $gid – the user ID and group ID of the owner of file.

$size – size of the file in bytes.

$atime, $mtime, $ctime – the file Timestamps. They are the last access time, time of last change, and last modification time, respectively.

$blksize, $blocks – the system blocksize and the number of blocks that make up the file.

More on the “mode” returned by the stat and lstat commands

* my ($dev, $ino, $**mode**, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize, $blocks) = **stat** ($file);

* $mode – the nine least significant bits in the $mode are the set of permission bits for the file.

* Upper bits identify file type.

Use bitwise operators to work with the file “mode”

* Bitwise operators work in a bit-by-bit manner

* & Bitwise And -- which bits are true in both arguments
* | Bitwise Or -- which bits are true in either argument
* ^ Bitwise XOR -- which bits are true in either argument but not both
* << Bitwise shift left shift left argument by number of bit shown in right argument; add zero-bits at the least significant bits
* >> Bitwise shift right shift left argument by number of bits shown in right argument; discard the least significant bits

Bitwise AND and Bitwise OR

Bitwise AND -- &

```
  1010
&1001
-------
  1000
```

Bitwise OR -- |

```
   10011
|  10110
-----------
   10111
```

Bitwise Exclusive OR (xor) and Bitwise shift left

Bitwise Exclusive OR-- ^

```
  1010
^1001
-------
  0011
```

Bitwise shift left -- <<

```
    100
<< 010
----------
 10000
```

Bitwise shift right

Bitwise shift right -- >>

```
  10110
>>00010
---------
  101
```

Working with File Timestamps

* Timestamps are the 9th, 10th, and 11th values in the list of values returned by **stat** and **lstat**.

* Convert timestamps to human-readable form with the **localtime** function.

my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime, $ctime, $blksize, $blocks) = **stat** ($file);

my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst) = **localtime** ($atime);

Time Stamp Exercise

* 1. Create a program that prints out the name of a file along with the time, day, month, and year the file was created.

Use bitwise operators to work with the file “mode”

* $mode – the nine least significant bits in the $mode are the set of permission bits for the file.

* -rwxr--r-- corresponds to octal 0744
* 0744 equals binary 0111100100

```
my ($dev, $ino, $mode, $nlink, $uid, $gid, $rdev, $size, $atime, $mtime,
$ctime, $blksize, $blocks) = stat ($file);

if ($mode & 0001) {
        print "$file is world executable.\n";
}
```

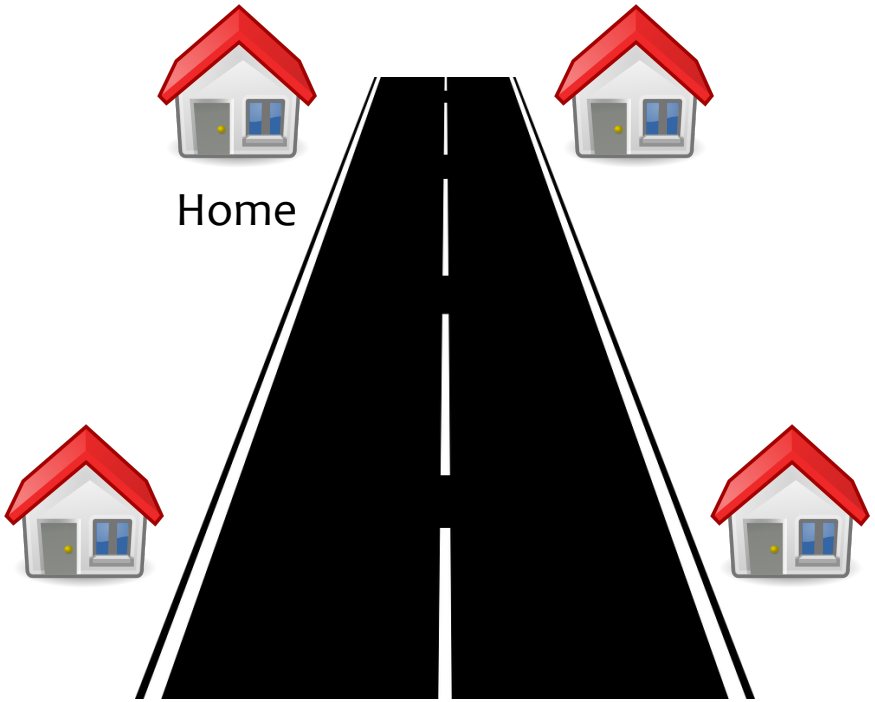
Directory Operations with Perl

Analogy:

Street = File System

House = Current Directory

Rooms = Subdirectories



Directory Operations

* Current Working Directory (CWD) – the directory your program starts in.
  getcwd(); # Tells you your CWD. use Cwd;

* chdir – changes current working directory
  chdir "$newDir" or die "Couldn't change to $newDir: $!";

* All your program's operations will be from your CWD

```
open $file, "testFile";                  # opens testFile in CWD
open $file, "/tmp/testFile";             # Full pathname opens testFile in /tmp
chdir '/tmp';  open $file, "testFile"    # opens testFile in /tmp
```

How to See Files in Directories

* See files in directories by:

- **Globbing** – expanding patterns (e.g. * or [abc]*) into filenames
  - glob '*' # the glob operator
  - <*> # the old glob operator

- **Directory Handles**
  - Like file handles, except used on directories

  - opendir $dh, $whateverDir or die "Couldn't open $whateverDir: $!";
  - while (readdir $dh) { print "$_\n"; }
  - closedir $dh;

Things to Do with Files

* Rename them

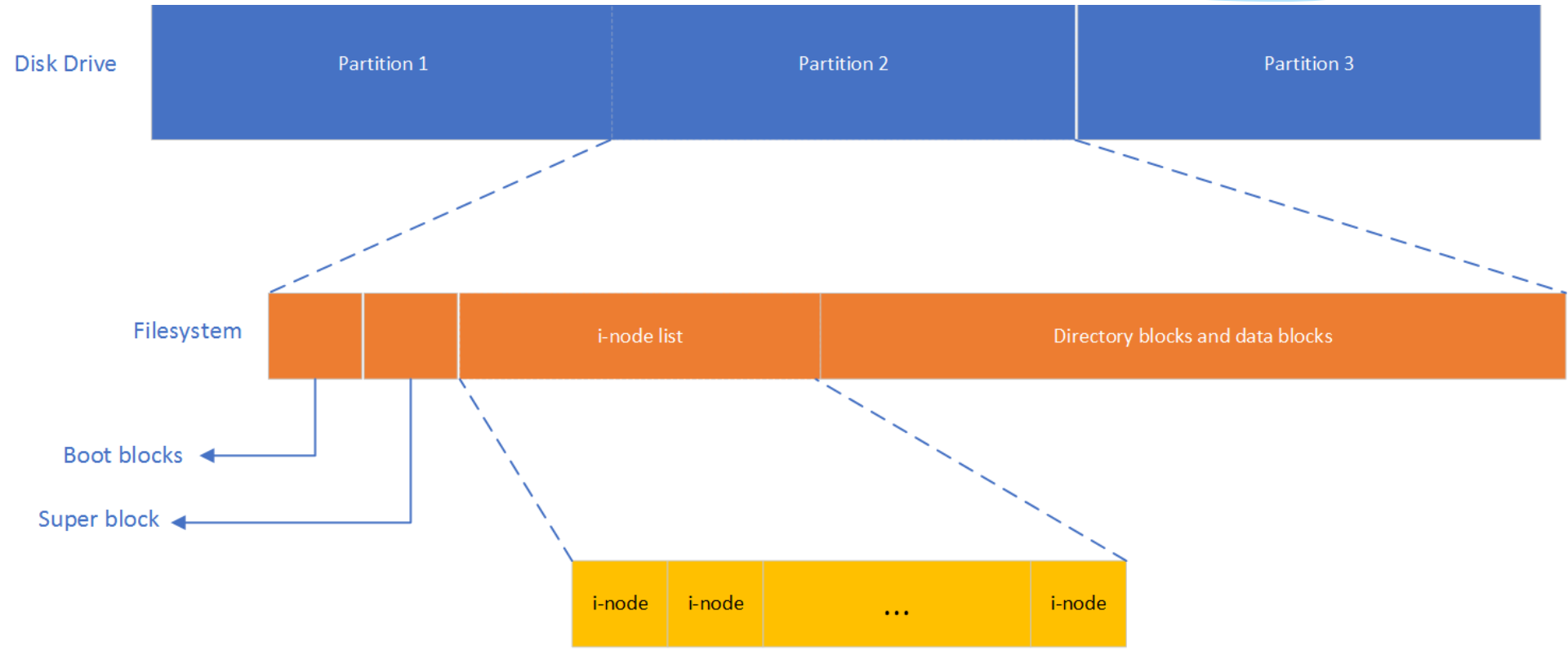
  rename ‘originalFileName’, ‘newFileName’;

* Delete them

  unlink ‘fileName’;

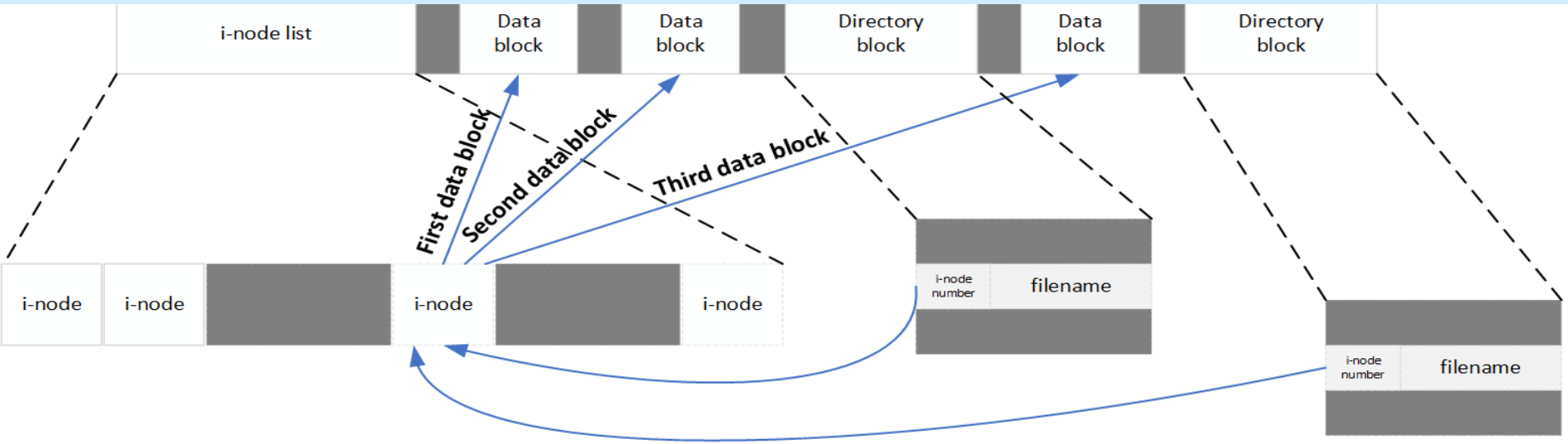
  unlink glob ‘*.tar’;

File Systems Explained



Simulacrum of disk drive, partitions and a filesystem from the book, "Advanced Programming in the UNIX Environment" by W. Richard Stevens

File Systems Explained, CONT'D



Simulacrum of "Filesystem in more detail"

Creating Links to files

* link ‘originalFile’, ‘linkToFile’;
  * link doesn’t work across file systems
* symlink ‘originalFile’, ‘softLinkToFile’;
  * Works across file systems and with directories

Reading softlinks, making directories, and removing directories

* $realFile = readlink ‘whateverFile’;

    # Gives the contents of the symlink

* mkdir ‘testDir’, 0755 or die “Couldn’t create directory, testDir: $!”;

    # Creates the “testDir” directory with the given permissions

* rmdir ‘testDir’;

    # Removes directory. Directory must be empty.

Changing Permissions

* chmod 0744, ‘testFile’;

    # Modifies the permissions of the file(s) or directories

Recursion

* Directory structure
  * topDir → middleDir → bottomDir

Stack

---

  * printDirStructure(topDir);
  * printDirStructure(middleDir);
  * printDirStructure(bottomDir);

Exercise 1

* Make a copy of the “printRelativeDirStructure.pl” program and rename it (e.g. call it ‘printRelativeFilePaths.pl’). Then modify that copy so that it prints all of the files in the “testDirs” directory structure.

* Challenge Program: Make a copy of the new program you just created. Perhaps rename it to ‘countFiles.pl’. Modify the copy so that it prints out the total number of files in the testDirs directory structure.

Exercise 2

* Write a program that goes through the testDirs directory structure and changes the permissions of every .csv file to 0664.

Exercise 3

* Re-write the find.pl program so that it uses **glob** instead of **opendir** and **readdir**.

Introduction to References

* How could you store the following data in a data structure?

* NFL
  * AFC
    * AFC-East → Patriots, Bills, Dolphins, Jets
    * AFC-West → Chiefs, Chargers, Raiders, Broncos
    * AFC-North → Steelers, Ravens, Bengals, Browns
    * AFC-South → Jaguars, Titans, Texans, Colts
  * NFC
    * NFC-East → Eagles, Cowboys, Redskins, Giants
    * NFC-West → Rams, Seahawks, Cardinals, 49ers
    * NFC-North → Vikings, Lions, Packers, Bears
    * NFC-South → Saints, Panthers, Falcons, Buccaneers

References

* References are values that *refer*, i.e. point to a data structure (e.g. an array or hash).

* References are similar to “pointers” in the C programming language.

* References can be stored wherever a scalar value can be stored, e.g. in
  - scalar variables
  - Arrays
  - hashes

How to create a Reference

* Create a reference by placing a backslash immediately before an array or hash. For example:
  * \@names -- This creates a reference to the array '@names'
  * \%loonyTunes– This creates a reference to the hash, '%looneyTunes'

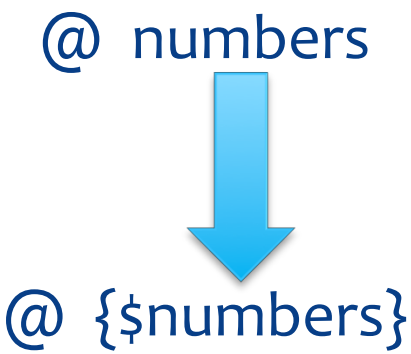
How to Assign References

```
my @grades = qw( 97 83 100 62 78 85 72 93 89 65 );     # create array

my $grades = \@grades;                                  # create and assign reference
```

How to Dereference

* @numbers = (1..10);
* $numbers = \@numbers;



How to Dereference

* @numbers = (1..10);
* $numbers = \@numbers;

$ numbers [2]

$ {$numbers} [2]

Dereferencing references

```
my @grades = qw( 97 83 100 62 78 85 72 93 89 65 );     # create array

my $grades = \@grades;                                  # assign reference

if (@{ $grades } == @grades) {
    print “The first grade is ${ $grades }[0]\n”;   # dereference reference
}
```

Dereferencing references

* my @numbers = (1..100);
* my $arrayRef = \@numbers;

* print “$numbers[3]\n”; # This is the same as below
* print “${ $arrayRef }[3]\n”; # This is the same as above

* print “$$arrayRef[3]\n”; # Drop curly braces for simple scalar ref

Why Use References?

* Save memory space when passing to subroutines
* Cleanly work on arrays and hashes in subroutines
* Build multi-level data structures

Nested References

* @secEast = (“Florida”, “Georgia”, “Kentucky”, “Missouri”, “South Carolina”, “Tennessee”, “Vanderbilt”);

* @secWest = (“Alabama”, “Arkansas”, “Auburn”, “LSU”, “Mississippi State”, “Ole Miss”, “Texas A&M”);

* @sec = (\@secEast, \@secWest);

* ${$sec[0]}[0] # What does this refer to?

Nested References CONT'D

* @big10East = ("Penn State", "Ohio State", "Michigan", "Indiana", "Maryland", "Michigan State", "Rutgers");

* @big10West = ("Wisconsin", "Iowa", "Nebraska", "Minnesota", "Northwestern", "Illinois", "Perdue");

* @big10 = (\@big10East, \@big10West);

Dereferencing Nested References

@d1 = (\@sec, \@big10, \@acc);

* $d1[0]; # Points to \@sec reference
* ${$d1[0]}[0]; # Points to \@secEast reference
* ${${$d1[0]}[0]}[0] # Points to “Florida”

Simplifying Nested Dereferencing

* ${REFERENCE}[$x] can be re-written as REFERENCE→[$x]
    * So,
* ${$anyRef[0]}[0] == $anyRef[0]→[0]
* ${${$anyRef[0]}[0]}[0] == $anyRef[0]→[0]→[0]

Simplifying Nested Dereferencing

@d1 = (\@sec, \@big10, \@acc);

* $d1[0]; # Points to \@sec reference

* ${$d1[0]}[0]; # Points to \@secEast reference
* $d1→[0] # Points to \@secEast reference

* ${${$d1[0]}[0]}[0] # Points to “Florida”
* $d1[0]→[0]→[0] # Points to “Florida”

Further Simplifying Nested Dereferencing

@d1 = (\@sec, \@big10, \@acc);

* $d1[0]; # Points to \@sec reference

* ${$d1[0]}[0]; # Points to \@secEast reference
* $d1→[0] # Points to \@secEast reference

* ${${$d1[0]}[0]}[0] # Points to “Florida”
* $d1[0]→[0]→[0] # Points to “Florida”

* $d1[0][0][0] # Points to “Florida”

Why Use References?

* Space saving when passed to subroutines
* Change of original data structure in subroutines
* Easy to build complex (nested) data structures

References to Hashes

* %family= (
        dad => 43
        mom => 41
        sally => 18
        joe => 16
        john => 14
        josephene => 12
    );

$family_ref = \%family;

Anonymous Array References

Create anonymous array reference with square brackets [].

Example:

```
@numbers = (1..10);
$ref = \@numbers;
```

OR you could do this:

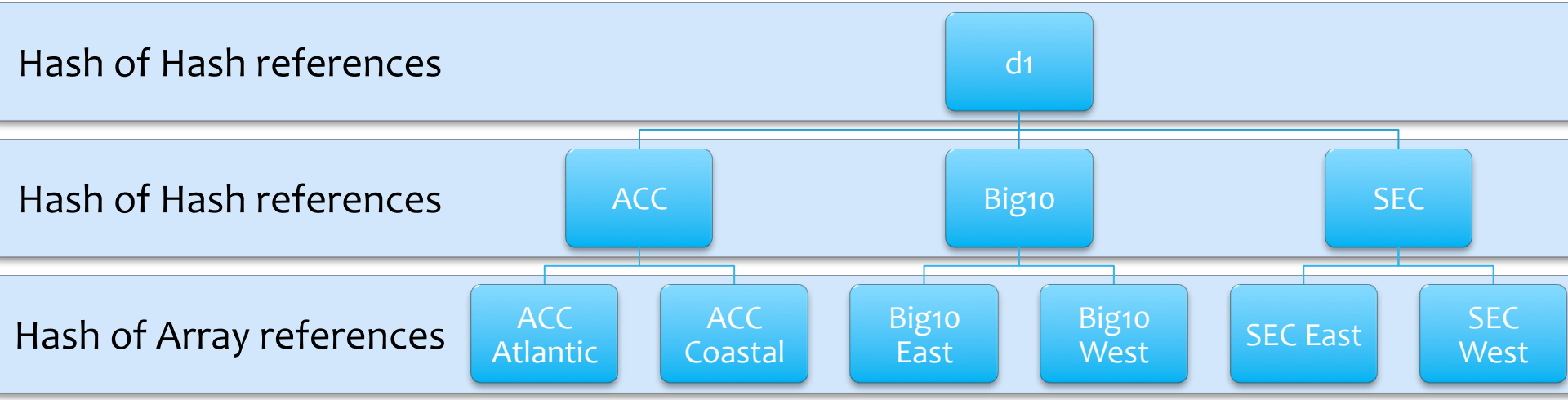
```
$ref = [ 1..10 ];
```

Anonymous Hash References

```
%hash = (
        Bugs => ‘Bunny’,
        Tasmanian => ‘Devil’,
        Elmer => ‘Fudd’,
);
$hashRef = \%hash;

OR
$hashRef = { Bugs => ‘Bunny’, Tasmanian => ‘Devil’, Elmer => ‘Fudd’};
```

d1football.pl program with Hashes



Array Values ----------→

Penn State
Ohio State
Michigan
Indiana
Maryland
Etc.

Florida
Georgia
Kentucky
Missouri
South Carolina
Etc.

d1football.pl program with Hashes

```
my %d1;

my @secEast = (“Florida”, “Georgia”, “Kentucky”, “Missouri”, “South Carolina”,
“Tennessee”, “Vanderbilt”);

my @secWest = (“Alabama”, “Arkansas”, “Auburn”, “LSU”, “Mississippi State”, “Ole Miss”,
“Texas A&M”);

my %sec = (“SEC East” => \@secEast, “SEC West” => \@secWest);

$d1{SEC} = \%sec;
```

Auto-Vivification

* Auto-Vivification – Perl makes arrays and hashes “come alive” on the fly, as we need them.

* When we dereference a non-existent variable to get to a non-existent array or hash, Perl automatically creates the data structure.

d1football.pl program with Hashes

```
my %d1;

Conference,Big10
Division,Big10 East
Teams,"Penn State","Ohio State","Michigan","Indiana","Maryland","Michigan State","Rutger
Division,Big10 West
Teams, "Wisconsin","Iowa","Nebraska","Minnesota","Northwestern","Illinois","Perdue";

$d1{'Big10'}{'Big10 East'}{Teams}[0] = "Penn State";
```

d1football.pl program with Hashes

```
my %d1;

Conference,Big10
Division,Big10 East
Teams,"Penn State","Ohio State","Michigan","Indiana","Maryland","Michigan State","Rutger
Division,Big10 West
Teams, "Wisconsin","Iowa","Nebraska","Minnesota","Northwestern","Illinois","Perdue";
```

Locating a substring with **index**

The **index** function returns the location of a substring in a larger string.

Syntax: index (“$bigString”, “$smallerString”);

```
$first_s = index (“some string”, “s”);
    # $first_s is assigned the value 0

$second_s = index(“some string”, “s”, $first_s + 1);
    # $second_s is assigned the value of 5
```

index and **rindex**

* **index**'s third argument starts its search past the start of the string
* $string = "friendly freckled face";

```
$place = index ($string, "f");                  # $place == 0
$place = index ($string, "f", $place + 1);      # $place == 9
$place = index ($string, "f", $place + 1);      # $place == 18
$place = index ($string, "f", $place + 1);      # $place == -1
```

* The **rindex** function begins its search from the end of a string.

* $string = "the end of time";
* $when = rindex ($string, "e"); # $when == 14

Exercise 1

* Write a program that states where each occurrence of the letter ‘i’ is in the word:

  “Supercalifragilisticexpialidocious”

The program should use a while loop to state the location of each letter ‘i’ in ‘Supercalifragilisticexpialidocious’, except for the first occurrence. That first location can be given before the while loop is started.

Exercise 2

* Write a program that states where each occurrence of the letter ‘c’ is, from the last occurrence down to the first occurrence, in the word:

  “Supercalifragilisticexpialidocious”

Except for the first occurrence of the letter ‘c’, the program should use the **rindex** command a while loop to find the location of each letter ‘c’ in ‘Supercalifragilisticexpialidocious’. That first location can be given before the while loop is started.

Getting substrings with substr

* substr – takes 3 arguments: a string, a string position (like that returned from **index**), and a string length.

```
$string = 'Supercalifragilisticexpialidocious';
$subString= substr ($string, index($string, "c"), 4);
print "I'm goin' back to $subString!\n";
```

* Leave off 3rd argument to return from location to end of string

```
$last_c = rindex ($string, "c");
$ending = substr($string, $last_c);
print "Was the food atro$ending or deli$ending?\n";
```

Exercise 2

Write a working program that contains the lines of code below so we can see what the print statements print out.

```
$string = 'Supercalifragilisticexpialidocious';
$subString= substr ($string, index($string, "c"), 4);
print "I'm goin' back to $subString!\n";

$last_c = rindex ($string, "c");
$ending = substr($string, $last_c);
print "Was the food attro$ending or deli$ending?\n";
```

sprintf

* sprintf – just like printf except that it returns string instead of printing it to STDOUT.

* $amount = sprintf “%4.2f\n”, 3.251383994;
* print “Your total is \$$amount.\n”;

* See perlfunc manpage for details and list of formatting options

Exercise 3

* Write a program that takes the number in the string.txt file (available for download), and uses the **index**, **rindex**, and **substr** functions to get the number out of the string. The string is:

'abcdefghtuvwx3.14159265358979323846jklmnpqrsyz'

* For this program, let's say you know that the first and last digits in the number are 3 and 6, respectively.
* Then, using sprintf, round the number down to two digits after the decimal, format the number in a field of 5 spaces, and save it to a variable.
* Lastly, print the contents of the variable to the screen

Sorting

* The **sort** operator sorts in ascii order.
  * Numbers come before letters
  * Uppercase letters come before lowercase letters

```
@numbers = (1..40);
foreach (sort @numbers) {
        print “$_\n”;
}
```

Advanced Sorting

* Tell sort how to sort in a subroutine

```
sub by_num {
        if ($a < $b)
                { -1 }
        elsif ($a > $b)
                { 1 }
        else
                { 0 }
}
```

* Then place subroutine between **sort** and the list to be sorted:

```
        sort by_num @array
```

How Sort uses Subroutines

* sort
* Subroutine: by_num
* List of values: qw (3 11 22 4 5 7 8 41 15 2 12 13 10 14 6)

* Pair of values, e.g. 3 and 11

* Subroutine return values:
  * -1 → order of values stays the same, i.e. 3, 11
  * 1 → order of values is swapped, i.e. 11, 3
  * 0 → order of values stays the same, i.e. 3, 11

How sorting works

* if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }

* If we sort the numbers 2, 1, and 3
  * First 2 and 1 are compared. A 1 is returned.
  * Next, perhaps 2 and 3 are compared. A -1 is returned.
  * Now Perl knows the order: 1, 2, 3

How Descending sorting works

* if ($b < $a) { -1 } elsif ($b > $a) { 1 } else { 0 }

* If we sort the numbers 2, 1, and 3
  * First 2 and 1 are compared. A -1 is returned. 2 comes before 1.
  * Next, perhaps 2 and 3 are compared. A 1 is returned. 3 comes before 2.
  * Now Perl knows the order: 3, 2, 1

Advanced Sorting, CONT'D

* Numerical and string comparisons can be shortened:

* sub by_num {
  if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
* }

* The above subroutine is the same as:
* sub by_num { $a <=> $b} # <=> is the 'spaceship operator'

* The string version is:
  * sub by_char { $a cmp $b} # But this is the same as just using sort

Advanced Sorting, CONT'D

* Short subroutines can be typed inline:

```
foreach (sort { $a <=> $b } @numbers) {
    print "$_\n";
}
```

* Values are compared (not $a and $b), so descending sorts can be done like this:

```
sort { $b <=> $a }
```

Exercise 4

* Write a program that uses a sort subroutine to sort the list, 0..40, correctly.

* Sort the list from smallest to largest value and from largest to smallest.

Sorting Hashes by Value

* %grades = (
    “Johnny” => 83,
    “Jane” => 92,
    “Leanne” => 88,
    “Evan” => 90
);

To sort in ascending order, use: $grades{$a} <=> $grades{$b}

To sort in descending order, use: $grades{$b} <=> $grades{$a}

Sorting Hashes by Value

* %grades = (
    "Johnny" => 83,
    "Jane" => 92,
    "Leanne" => 88,
    "Evan" => 90
    "William" => 92,
);

If grades are the same, sort by name:

```
sub hash_sort {
    $grades{$a} <=> $grades{$b} or
        $a cmp $b
}
```

Exercise 5

* Sort the grades from the following hash in descending order:

* %grades = (
    "Johnny" => 83,
    "Jane" => 92,
    "Leanne" => 88,
    "Evan" => 90,
    "William" => 92,
    "Sharon" => 80,
);

If grades are the same, alphabetically sort by name:

Exercise 4

* Using find.pl as a template, do the following:
  * Locate the file named, “PG_IOPS.csv”, in the testDirs directory structure.
  * Open that file and match the lines that begin with "No.".
  * Strip out all of the double quotes from those lines.
  * Split the lines using commas (,) as the character to split on.
  * Add the values to an array, but only add values that start with a number or an uppercase ‘X’.
  * Use the **sort** command to sort the list of values and print them out to the screen.

Process Management

* Launching other programs from your Perl programs

* We'll cover 2 ways:
  * The ***system*** function, e.g. system 'ls';
  * Back quotes. For example, `who`;

The system function

* **system**
  * Creates a child process that runs the command you give **system**
  * Shares your program's STDIN, STDOUT, STDERR
  * Your program waits for child process to finish

  * Returns 0 if successful
  * Returns non-zero if it fails

Using Back Tics ``` to start a process

* Back Quotes or Back Tics can be used to start a command
* Instead of sending output to standard out, the outpout is returned and can be captured in a variable

```
$dateInfo = `date`;
print “The date information is: $dateInfo\n”;
```

Don’t use unless you’re going to capture output

Process FileHandles

open WHO, ‘who|’ or die “Couldn’t open ‘who’ command: $!”;
open CAT, ‘|cat > ./someFile’ or die “Couldn’t open ‘cat’: $!”;

* Also, there’s a 3-argument version:

open WHO, ‘-|’, ‘who’ or die “Couldn’t open ‘who’ command: $!”;
open CAT, ‘|-’, ‘cat > ./someFile’ or die “Couldn’t open ‘cat’: $!”;

Challenge Program

* Print out the directory structure contained in the ‘perfstats.zip’ file.
  * Unzip perfstats.zip
  * Recursively descend into the perfstats directory
  * Unzip each .ZIP file found. Each will create a new directory
  * Print out each directory discovered
  * After printing out all directories in a directory, zip the directory back up and return the environment back to its original state

Congrats!

* Your Perl foundation is complete
* You can write programs
* You can read programs
* You are now more skillful than you were before you took this course
* Use your Perl skills as often as possible