

# Programming in Perl

- Introduction
- Scalars
- Lists and Arrays
- Control Structures
- I/O
- Hashes
- Regular Expressions
- Dealing with Files
- Subroutines
- Directory and File Manipulation

# History

- Perl stands for Practical Extraction and Report Language.
- Created by Larry Wall in the mid 1980s.
- Released to Usenet readers and became popular.
- Perl is free for use and is distributed under the GNU public license.

# Advantages of Perl

- Fills the gap between programming in a conventional compiled language and shell programming.
- Is very high-level. A typical Perl program may take 30% to 70% as much code as a C program.
- Good for accomplishing quick tasks and primarily for text manipulation.

# Perl Is Interpreted

- Your Perl program is initially compiled into bytecodes when you invoke your program and these bytecodes are then interpreted.
- Similar in some ways to Java.
- Faster than shell interpretation.
- Still much slower than conventionally compiled programs.

# Sample Perl Program

- First line indicates the name of the program that executes the file. Can execute this file like a shell script. The `-w` means to print warnings.
- Second line is a comment.
- Third line is a pragma to indicate that variables should be declared and strings should be quoted.
- Fourth line prints the string.
- Last line exits the program.

```
#!/usr/bin/perl -w
# This is a hello world program.
use strict;
print "Hello world!\n";
exit 0;
```

# Basic Concepts

- No main function, but can have subroutines.
- Many features taken from C and shell commands.
- Easy to write a program with a few commands to perform simple tasks.

# Features Similar to C

- Many operators.
- Many control structures.
- Supports formatted I/O.
- Can access command line arguments.
- Supports standard input, output, and error.

# Features Similar to Shell Programming

- Comments: # to the end of the line
- \$variables
- Interpolation of variables in “strings”.
- Support for command line arguments.
- Implicit conversions between strings and numbers.
- Support for regular expressions.
- Some control structures.
- Many specific operators similar to shell commands or Unix utilities.

# Scalar Data

- Scalars represent a single value.
- Scalar types:
  - Strings
  - Numbers
- Strings and numbers, like in the shell, are used almost interchangeably in Perl.

# Numbers

- Perl stores all numbers as double-precision values internally.
- Numeric Literals
  - floating-point literals
  - integer literals
    - decimal integer literals
    - non-decimal integer literals

# Floating-Point Literals

- Floating-point literals (or constants) in Perl are similar to those in C.
- All of the following represent the same value.

149.567

149567e-3

1.49567E2

0.0149567e4

# Integer Decimal Literals

- Similar to C.

0    -54    511

- Can use underscores for large values.

2839683876

2\_839\_683\_876

# Integer Nondecimal Literals

- Similar to C.

0177           # literals beginning with zero are octal  
# constants

0x7f           # literals beginning with 0x are  
# hexadecimal constants

- Not found in C.

0b1111111    # literals beginning with 0b are binary

# Operators Similar to C

- assignment: =
- arithmetic: +, -, \*, /, %
- bitwise: &, |, ^, ~, <<, >>
- relational: <, <=, ==, !=, >=, >
- logical: &&, ||, !
- binary asg: +=, -=, \*=, ...
- Increment: ++, --
- Ternary: ?:

# Operators Different from C

- `**` # exponentiation
- `<=>` # numeric comparison
- `=~`, `!~` # match operators
- `x` # string repetition
- `.` # string concatenation
- `eq,ne,lt,gt,le,ge` # string relational
- `cmp` # string comparison
- `\,` `=>` # list

# Strings

- Unlike many conventional programming languages, string is a basic type in Perl.
- String Literals
  - single-quoted strings
  - double-quoted strings

# Single-Quoted Strings

- Use single-quoted strings when you do not want variables to be interpolated.
- Can use the `\` character to indicate that a single quote is part of the string (`'...\''`) or a backslash is part of the string (`'...\\'`).
- The `\` followed by any other character is just a regular `\`.

`'Hello World!'`

`'This is just a \ character.'`

`'Whalley\'s Class'`

`\"`

`'The \\ is used to access directories in DOS.'`

# Double-Quoted Strings

- Double-quoted strings are similar to C in that you can use the backslash to specify a special character.
  - “This line ends with a newline.\n”
  - “These\twords\tare\tseparated\tby\ttabs.”
  - “The \” title\” of a book should be quoted.”
  - “The price is \ \$1,000.”
- They can also be used to interpolate variables, as in the Bourne shell.

# String Operators

- '.' is used for string concatenation.
  - “One string can be concatenated ” . “with another.”
  - 'The price is \$1,000.' . “\n”
- 'x' is used for string repetition.
  - “double” x 2 eq “doubledouble”
  - “ ” x 10 . “means 10 blanks in a row.”

# Implicit Conversions between Strings and Numbers

- Implicit conversions are performed depending on the operator that is used. The coercions are performed without any warnings.

9 x "5"        # "99999"

"1" + "2"     # 3

"45" - 1.7    # "447"

# Scalar Variables

- Scalar variable names are preceded by '\$'. Unlike shell variables, a '\$' is always used.
- General form.  
 $\$[A-Za-z\_][A-Za-z\_0-9]^*$
- Scalars can hold both strings and numbers.

# Declaring Scalar Variables

- If you use the following pragma:

```
use strict;
```

then all variables must be declared. You can do this with the “my” operator.

- General form. Use the first form to declare one variable. Use the second form to declare multiple variables.

```
my <variable_name>;
```

```
my (<variable_name>, ..., <variable_name>);
```

- Variable declarations can go anywhere, but are often placed at the top of the program.

# Example Scalar Variable Declarations

```
my $sum;           # used to hold a sum of values
my ($i, $j, $k);  # counter variables
my $line;         # contains a line of text
my $n = 0;        # variable with an initial value
my $s = "";       # another variable with an initial value
my $a = $b;       # variables can be initialized to have a
                  # run-time value
```

# Interpolation of Variables in Strings

- Variables are interpolated inside double-quoted strings. Say the value of `$n` is 7. The string  
“The value of `\$n` is: `$n`.\n”

would be interpolated to be:

“The value of `$n` is: 7.\n”

- One can use the form: `${name}` when the variable is followed by a character in a string that could be part of an identifier. Say the value of `$day` is “Tues”. The string  
“Today is `${day}day`.\n”

would be interpolated to be:

“Today is Tuesday.\n”

# Assigning Scalar Values

- The assignment operator is '=', which is the same operator that is used in C.
- Scalars variables can be assigned numeric or string literals, other variables, or expressions consisting of operators, literals, and variables.

```
$m = 4;
```

```
$n = "ana";
```

```
$n = "ban" . $n;
```

```
$m += 1;
```

# Undef Value

- Variables have a special **undef** value before they are first assigned.
- A variable containing **undef** is treated as zero when it is used as a numeric value.

```
$sum += $n;
```

- A variable containing **undef** is treated as an empty string when it is used as a string value.

```
$s = $s . “.old” ;
```

# Print Operator

- The print operator can be used to print a list of expressions (strings, numbers, variables, or a combination of operands with operators). By default it prints to standard output.

- General form.

```
print [expression[, expression]*];
```

- Examples that all print the same output:

```
print "a=$a\n";
```

```
print "a=", $a, "\n";
```

```
print "a=", $a*1, "\n";
```

# Line Input Operator <STDIN>

- The <STDIN> operator can be used to read a line of input from standard input, up to and including the next newline character, into a string.

```
$line = <STDIN>;
```

- If the end-of-file is reached, then <STDIN> returns undef (or the empty string).

# The Chomp Operator

- The chomp operator can be used to remove a newline from the end of a string.

```
$line = <STDIN>;           # chomp after reading the  
chomp($line);              # line
```

```
$line = <STDIN>;           # (...) in function calls are  
chomp $line;               # not required
```

```
chomp($line = <STDIN>);    # can do both in one step
```

# String Relational Operators

- “eq”, “ne”, “lt”, “gt”, “le”, and “ge” are the string relational operators. The last 4 tests check the ASCII order, character by character.

\$answer eq “yes”

\$a lt \$b

100 lt 2

# would be true

# Size of a String

- You can determine the number of characters in a string by using the length function.

- General form.

`length <string>`

- Example:

```
print "The length of \"$s\" is", length s, ".\n";
```

# Basic Control Structures

- If-Elself-Else
- While
- Until
- For

# Boolean Conditions

- These control structures rely on boolean conditions. Numeric and string relational operations return a value that is treated as either true or false. What happens if you use a scalar value?
  - Undef is considered to be false.
  - Zero is false, all other numeric values are true.
  - The empty string and “0” are false, all other strings are true.

# If-Elsif-Else Control Structure

- General form. Note the {...} are required even if there is only one statement.

```
if (<boolean_condition>) {  
    <one_or_more_statements>  
}  
[elseif (boolean_condition) {  
    <one_or_more_statements>  
}]*  
[else {  
    <one_or_more_statements>  
}]
```

# If-Elself-Else Examples

- If example:

```
if ($n > $max) {  
    $max = $n;  
}
```

- If-Else example:

```
if ($a < $b) {  
    $max = $b;  
}  
else {  
    $max = $a;  
}
```

# If-Elsif-Else Examples (cont.)

- If-Elsif example:

```
if ($max < $n) {  
    $max = $n;  
}  
elsif ($min > $n) {  
    $min = $n;  
}
```

# If-Elself-Else Examples (cont.)

- If-Elself-Elself example:

```
if ($a == 1) {  
    print "one\n";  
}  
elseif ($a == 2) {  
    print "two\n";  
}  
elseif ($a == 3) {  
    print "three\n";  
}
```

# If-Elsif-Else Examples (cont.)

- If-Elsif-Else example:

```
if ($answer eq "yes" || $answer eq "Yes") {  
    $n = 1;  
}  
elsif ($answer eq "no" || $answer eq "No") {  
    $n = 0;  
}  
else {  
    print "Invalid answer.\n";  
}
```

# Defined Function

- Can use the defined function to see if a value has not been assigned a value.

```
# enter the if statement if $n has been assigned a value
if (defined($n)) {
    ...
}
```

# While Control Structure

- Performs one or more statements while a condition is true.
- General form. Again the {...} are required.

```
while (<boolean condition>) {  
    <one_or_more_statements>  
}
```

# While Examples

```
# echos the input
```

```
while (defined($line = <STDIN>)) {  
    print $line;  
}
```

```
# prints squares of the values 1 to 100
```

```
$i = 1;  
while ($i <= 100) {  
    print $i**2;  
    $i++;  
}
```

# Until Control Structure

- Performs one or more statements until a condition is true (i.e. while a condition is false).
- General form. Again the {...} are required.

```
until (<boolean_condition>) {  
    <one_or_more_statements>  
}
```

# Until Examples

```
# echos the input
```

```
until (!defined($line = <STDIN>)) {  
    print $line;  
}
```

```
# prints squares from 1 to 100
```

```
$i = 1;  
until ($i == 101) {  
    print $i**2;  
    $i++;  
}
```

# For Control Structure

- The for control structure is similar to the for statement in C.
- General form. Again the {...} are required.

```
for (initialization; test; increment) {  
    <one_or_more_statements>  
}
```

# For Examples

```
# print 100..1 on separate lines
for ($i=0; $i < 100; $i++) {
    print 100-$i, "\n";
}
```

```
# read n and print summation of 1..n
chomp($n = <STDIN>);
$sum = 0;
for ($i=1; $i <= $n; $i++) {
    $sum += $i;
}
print "summation of 1..$n is $sum.\n";
```

# For Examples (cont.)

```
# infinite loop (no condition means the condition  
# defaults to be true each time)  
for ( ; ; ) {  
    ...  
}
```

# Lists and Arrays

- A list in Perl is an ordered collection of scalar data.
- An array in Perl is a variable that contains a list.
- Each element of a list can contain an independent scalar value, which can be a number or a string.

# List Literals

- Can represent a list of values in Perl.

- General form.

( <scalar\_value>, <scalar\_value>, ..., <scalar\_value> )

- Examples:

(1, 3, 5) # three numbers

("cat", "dog") # two strings

(1, "cat", 0.5) # can mix numbers and strings

(0, \$a, \$a+\$b, 0) # some values can be determined at  
# run-time

() # can have an empty list

# The *qw* Shortcut

- Can use the *qw* shortcut to create a list literal of *quoted words*.

```
# list literal below contains strings representing fruit  
( "orange", "apple", "pear", "lemon", "grape" )
```

```
# below is a similar assignment, but requires fewer chars  
qw/ orange apple pear lemon grape /
```

```
# can use other delimiters besides '/'  
qw! orange apple pear lemon grape !
```

```
# can use delimiters with "left" and "right" characters  
qw( orange apple pear lemon grape )  
qw< orange apple pear lemon grape >
```

# List Literals (cont.)

- Can use the range (..) operator to create list values by counting from the left scalar to the right scalar by ones.

- Examples:

(1..4)           # same as (1, 2, 3, 4)

(1.1..4.4)       # same as (1..4) since range values  
                  # have to be integers

(4..1)           # empty list since left value must be  
                  # greater than the right value

(1, 4..6, 10)    # can be used along with explicit list values

(\$m..\$n)        # range values can be determined at run time

# Array Variables

- Arrays are declared using the '@' character.
- General form. Note that the size of the array is not specified.

```
my @arrayname;
```

- Examples:

```
my @a;      # array a
```

```
my @nums;  # array of numbers
```

```
my @strings; # array of strings
```

# Array vs. Scalar Names

- Easy way to remember names:
  - \$ looks like an S: \$scalar
  - @ looks like an a: array
- Scalar and array names are in different name spaces. Could reuse the same names, but it is not recommended.
  - \$b = \$b[0];   # Assigns array element \$b[0] to scalar \$b
  - # The above code is confusing!

# Accessing Array Elements

- Accessing array elements in Perl has similar syntax to accessing array elements in C.
- General form. The '\$' is used since you are referring to a specific scalar value within the array. The expression is evaluated as an integer value. The first index of every array is zero.

`$arrayname[<expression>]`

# Examples of Accessing Array Elements

```
$a[0] = 1;           # can assign numeric constants
$s[1] = "Report";  # can assign string literals
print $m[$i];      # can index using a scalar variable
$a[$i] = $b[$i];   # can copy one element to another
$a[$i] = $a[$j];   # another example
$a[$i+$j] = 0;     # can index by an expression
$a[$i]++;          # incrementing $a[$i] by one
```

# Assigning List Literals

- Can assign list literals to arrays or scalars.

```
($a, $b, $c) = (1, 2, 3);           # $a=1; $b=2; $c=3;
($m, $n) = ($n, $m);              # can perform swaps
@nums = (1..10);                   # Can update entire arrays.
                                   # $nums[0]=1; $nums[1]=2; ...
($x, $y, $z) = (0, 1);            # $x=1; $y=2; $z=undef;
@t = ();                           # array with no elements
($a[0], $a[1]) = ($a[1], $a[0]);  # another swap
@fruit = ("orange", "apple");     # fruit has two elements
@fruit = qw/ orange apple /;     # similar assignment
```

# Accessing Entire Arrays

- Entire arrays can sometimes be accessed. Use `@arrayname` instead of `$arrayname[...]`.

```
@x = @y;           # copy array y to array x
@y = 1..1000;      # range oper does not have to
                  # be inside parentheses
@lines = <STDIN>;  # read all lines of input
                  # $lines[0]=<STDIN>;
                  # $lines[1]=<STDIN>;
                  # ...
print @lines;     # print all array elements
```

# Printing Entire Arrays

- Can print an entire array at once.

```
@fruit = ( "apple", "orange", "pear" );
```

```
print @fruit, "\n"; # prints "appleorangepear"
```

- Can print all array elements separated by spaces.

```
print "@fruit\n"; # prints "apple orange pear"
```

# Using the Array Name in a Scalar Context

- Using the array name when assigning it to a scalar or with a scalar operator results in the number of values being returned. It will not give a warning.

```
@array1 = ("cat", 2, "dog", 1, "hamster", 3);
```

```
@array2 = @array1;    # copies array1 to array2
```

```
$m = @array2;        # $m = 6;
```

```
$n = $m + @array2;   # $n = 12;
```

# Using a Scalar in a List Context

- Assigning a scalar to an array will result in the array containing a one element list.'

```
$m = 1;
```

```
@array = $m;      # @array = ( 1 );
```

```
@fruit = "apple"; # @fruit = ( "apple" );
```

```
@array = undef;   # @array = ( undef );
```

```
@array = ( );     # Really empties the array.
```

# Size of Arrays

- Perl arrays can be of arbitrary size, provided there is enough memory to hold it. The number of elements can vary during run-time.

```
my @fruit;           # at this point @fruit has no elements
...
$fruit[0]=" apple" ; # now @fruit has one element
$fruit[1]=" orange" ; # now @fruit has two elements
$fruit[99]=" mango" ; # now @fruit has 100 elements
                    # $fruit[2]..$fruit[98] have undef values
```

# The Last Element Index

- `$#arrayname`  contains the current last element index, which is one less than the number of elements.

`# Can be used to iterate through the array elements.`

```
for ($i=0; $i <= $#fruit; $i++) {  
    print $fruit[$i], "\n";  
}
```

`# Can be used to resize an array.`

```
$a[99] = $i;    # assigns value to 100th element of @a
```

...

```
 $#a = 9;      # now @a has only 10 elements
```

# Using Negative Array Indices

- Can use negative array indices to access elements from the end of the array.

```
print $a[-1];      # print the last element of @a
```

```
                  # similar to using $a[$#a]
```

```
print $a[-2];      # print the 2nd to last element of @a
```

# Push and Pop Operators

- Arrays are often used like a stack, so there is support for push and pop operations.
- The push operator takes two arguments:
  - an array
  - value to be pushed, which can be a list value
- The pop operators takes one argument:
  - an array

# Push and Pop Examples

```
push @nums, $i; # same as “$nums[++$#nums] = $i;”
push @a, “end” # adds “end” as a new element
push(@a, 1..5) # assigns 1..5 as 5 new elements
push(@a, @b) # adds the elements of @b at the end of @a
push @a, (1, 2, 3) # adds 1..3 as new elements to the end of @a
print pop @a; # same as “print $a[$#a]; $#a -= 1;”
pop @a; # same as “$#a -= 1;”
push @b, pop @a; # pops $a[$#a] and pushes it onto @b
@a = (); # makes @a become empty
$b = pop @a; # $b now contains undef
```

# Shift and Unshift Operators

- The shift and unshift operators are analogous to the pop and push operators, except they work on the first instead of the last element.
- Shift, like in the shell for the command line arguments, returns the first element of an array and shifts the other elements over to the *left*.
- Unshift adds a value to an array by shifting the current elements to the *right* and assigning the new value to the first element.

# Shift and Unshift Examples

```
@a = ("cat", 4, "dog");    # @a now has 3 elements
$b = shift @a;           # $b == "cat" &&
                           # @a == (4, "dog")
$c = shift @a;           # $c == 4 && @a == ("dog")
$d = shift @a;           # $d == "dog" && @a == ()
$e = shift @a;           # $e == undef && @a == ()
unshift @a, 1;           # @a == (1)
unshift @a, ("cat", "dog") # @a == ("cat", "dog", 1)
```

# Foreach Control Structure

- The foreach control structure is used to process an entire array or list.
- General form. The \$scalar gets assigned one value of the list or array for each iteration.

```
foreach $scalar (<list_or_array>) {  
    <one_or_more_statements>  
}
```

# Foreach Examples

```
# prints each element of the array nums, one per line
foreach $num (@nums) {
    print $num, "\n";
}
```

```
# pushes the items in the list onto the fruit array
foreach $item (qw/ apple orange pear grape /) {
    push @fruit, $item;
}
```

# Perl's Default Variable

- `$_` is Perl's default variable and is used as a shortcut to reduce the number of characters typed. It is used as a default when reading input, writing output, and as a default for the `foreach` control structure.

```
while (<stdin>) {    # Reads into $_ by default.  
    print;          # Prints from $_ by default.  
}
```

```
$sum = 0;  
foreach (@nums) { # Assigns to the $_ by default.  
    $sum += $_;  
}
```

# Input from the Diamond Operator

- Reading input from the `<>` operator causes programs to read from standard input when there are no command line arguments or from files specified on the command line.
- Allows Perl programs to mimic the behavior of Unix utilities. One difference is that the list of files specified on the command line are treated as a single file that is concatenated together.

# Example of Input from $\langle \rangle$

# mimics the *cat* Unix utility

```
while ($line= $\langle \rangle$ ) {  
    print $line;  
}
```

# can invoke by redirecting from standard input

```
cat.pl < input.txt
```

# can invoke by passing arguments on the command line

```
cat.pl input1.txt input2.txt input3.txt > output.txt
```

# The @ARGV Array

- The @ARGV array contains the strings representing the command line arguments at the start of the execution.
- Can process other command line options by shifting them from the @ARGV array before the first <> operation is performed.
- Note that \$ARGV[0] contains the first command line argument, not the name of the Perl file being invoked.

# Examples of Using the ARGV Array

```
# mimics the Unix echo utility
foreach (@ARGV) {
    print “$_ ”;
}
print “\n”;
```

```
# count the number of command line arguments
$i = 0;
foreach (@ARGV) {
    $i++;
}
print “The number of arguments is $i.\n”;
```

# Loop Control Operators

- Perl has three loop control operators.
  - last: used to break out of a loop
  - next: used to goto the next iteration
  - redo: used to repeat the current iteration

# Last Operator

- The last operator breaks out of the innermost loop in which it is contained. This is similar to the break statement in C.

```
# Sums the first 100 numbers read or entire input if less.
```

```
$i = 1;
```

```
$sum = 0;
```

```
while ($num = <STDIN>) {
```

```
    chomp($num);
```

```
    $sum += $num;
```

```
    if ($i++ == 100) {
```

```
        last;
```

```
    }
```

```
}
```

# Next Operator

- The next operator skips over the rest of the loop body and continues with the next iteration. This operator is similar to the continue statement in C.

```
# sums the positive elements of the array vals
$sum = 0;
foreach $val (@vals) {
    if ($val <= 0) {
        next;
    }
    else {
        $sum += $val;
    }
}
```

# Redo Operator

- The redo operator will go back to the top of the loop block, but without performing the increment portion, testing the loop condition, or advancing to the next value in the list.

```
foreach $s (@strings) {  
    print "Do you wish to print $s?\n";  
    my chomp($ans = <STDIN>);  
    if ($ans eq "yes") {  
        print $s, "\n";  
    }  
    elsif ($ans ne "no") {  
        print "'$ans' is not a valid answer.\n";  
        redo;  
    }  
}
```

# Reverse Operator

- The reverse operator takes a list or array of values as input and creates a new list with the values in reverse order.

```
@nums = 1..100;           # @nums = (1, 2, ..., 100);  
@revnums = reverse @nums; # @revnums = (100, 99, ..., 1);  
@revnums = reverse 1..100; # @revnums = (100, 99, ..., 1);  
@nums = reverse @nums;   # reverses @nums itself
```

# Reverse Operator in Scalar Context

- The reverse operator can be used in either an array or scalar context. In a scalar context it returns a reversed string after concatenating all of the strings in the list.

```
@animals = qw/ dog cat cow /;  
@backwards = reverse @animals;      # (“cow”, “cat”, “dog”)  
$backwards = reverse @animals;     # “woctacgod”  
$backone = reverse ($animals[1]);  # “tac”  
@nums = (10, 9, 23);  
$s = reverse @nums;                # ?
```

# Sort Operator

- The sort operator takes a list or array of values as input and creates a sorted list in ASCII order.

```
@fruit = qw( apple orange grape pear lemon );
@sortedfruit = sort @fruit; # (apple grape lemon orange pear)
print “ @sortedfruit\n”;   # prints sortedfruit on one line
foreach $f (sort @fruit) { # prints each fruit in sorted order
    print $f, “\n”;        # one per line
}
```

```
@nums = sort 98..101;      # assigns (100, 101, 98, 99)
$n = sort 98..101;        # assigns undef
```

# Hashes

- A hash is similar to an array in that individual elements are accessed by an index value and may have an arbitrary number of values.
- A hash differs from an array in that the indices are strings, which are called keys.
- The elements of a hash have no particular order.
- The hash contains key-value pairs. The keys have to be unique, but the values may not.
- A hash can be viewed as a very simple database, where a scalar data value can be filed for each key.

# Why Use a Hash?

- There are often relationships between sets of data that need to be maintained. You would like to efficiently access one set of data by using the key from another.
- Examples
  - word => meaning
  - student ID => name
  - loginname => name
  - employee ID => salary
  - title => author
  - barcode => price

# Hash Declarations

- Use the '%' preceding a name to identify a hash.  
my %book;  
my %products;
- The names of hashes are kept in a separate namespace from scalars and arrays. However, it is good practice to use a unique name for each hash.

# Hash Element Access

- General form. Use '\$' before the hashname to access an individual scalar value from a hash. Use '{' '}' instead of '[' ']' so that Perl will know it is a hash element instead of an array element being accessed.

`$hashname{$keyvalue}`

- If the `$keyvalue` contains a number or an expression, then the value is converted to a string, which is input to the hash function.

# Hash Element Access Examples

```
$names{67415} = "Doe, John";           # storing a name
$name = $names{67415};                 # name overwritten
$name = $names{46312};                 # retrieving a name
$name = $names{46312};                 # invalid key returns
                                        # an undef value

$name = $names{67415};                 # storing another name
foreach $id (@student_ids) {          # for each id
    print "$id=$names{$id}\n";         # print id=name
}
```

# Referring to the Entire Hash

- Use the '%' character to refer to the entire hash.

```
%new_hash = %old_hash;      # copy an entire hash
```

```
# initialize a hash by specifying key-value pairs
```

```
%fruit = ( "apple", 0.30, "orange", 0.45, "pear", 0.50);
```

```
# can use '=>' instead of a ','
```

```
%fruit = ("apple" => 0.30, "plum" => 0.45, "pear" => 0.50);
```

```
# cannot print an entire hash directly
```

```
print "%fruit\n";          # prints "%fruit"
```

```
# can turn a hash back into an array of key-value pairs
```

```
@fruitarray = %fruit;
```

# Keys and Values Function

- The keys function takes a hashname as input and creates a list of the current keys in the hash.
- The values functions takes a hashname as input and creates a list of the current values in the hash.

# hash initialization

```
%fruit = (“apple” => 0.30, “plum” => 0.45, “pear” => 0.50);
```

```
@k = keys %fruit;    # “apple”, “plum”, “pear” in some order
```

```
@v = values %fruit; # 0.30, 0.45, 0.50 in some order
```

# Each Function

- The each function takes a hash name as input and returns a two element list (key-value pair) for each iteration of a loop.

```
# print the name and price of each type of fruit
while ( ($name, $price) = each %fruit) {
    print “ $name = $price\n” ;
}
```

# Exists Function

- The exists function checks if a key exists in a hash. Note this function returns a true or false value, not the value associated with the key.

```
if (exists $fruit{$f}) {  
    print "The price of an orange is $fruit{$f}.\n";  
}
```

# Delete Function

- The delete function removes a key-value pair from a hash.

```
# hash initialization
```

```
%fruit = (“apple” => 0.30, “plum” => 0.45, “pear” => 0.50);
```

```
delete $fruit{“plum” };      # deletes “plum” => 0.45
```

```
@fruitarray = %fruit;      # assign to an array
```

```
print “@fruit\n”;          # only two key-value pairs will be
```

```
# printed
```

# Formatted Output with Printf

- The Perl printf function, unlike the print function, takes a format string as its first argument. Typically only used to print scalars.
- The format string has similar conversions as the C printf function.
- This feature should be used when you want more control over how the output should appear.

%s: string

%d: truncated decimal

%f: float

# Example Printf's

```
printf “ %7d\n” , $i;      # Prints integer value of $i right
                           # justified in 7 columns on one line.
printf “ %-10s” , $s;     # Prints $s as a left justified string in
                           # 10 columns.
printf “ %6.2f” , $f;     # Prints $f in a 6 column field with 2
                           # digits after the decimal point (ddd.dd).
printf “ %${max}s” , $s;  # Prints $s as a right justified string in a
                           # field that is $max columns wide.
                           # Note the use of the { }.
printf “ %s=%d\n” ,      # Prints $name as a string, followed by
$name, $val;            # an '=', followed by $val as an integer.
```

# Perl Regular Expressions

- Unlike most programming languages, Perl has built-in support for matching strings using regular expressions called patterns, which are similar to the regular expressions used in Unix utilities, like grep.
- Can be used in conditional expressions and will return a true value if there is a match. Forms for using regular expressions will be presented later.
- Example:  

```
if (/hello/)      # sees if "hello" appears anywhere in $_
```

# Perl Patterns

- A Perl pattern is a combination of:
  - literal characters to be matched directly
  - '.' matches any single character but a newline
  - '\*' match the preceding item zero or more times
  - '+' match the preceding item one or more times
  - '?' match the preceding item zero or one times
  - '(' and ')' for grouping
  - '|' match item on the left or item on the right
  - [...] match one character inside the brackets

# Examples of Perl Patterns

<code>/abc/</code>	<code># abc</code>
<code>/a.c/</code>	<code># a, any char but newline, c</code>
<code>/ab?c/</code>	<code># ac or abc</code>
<code>/ab*c/</code>	<code># a, zero or more b's, c</code>
<code>/ab cd/</code>	<code># ab or cd</code>
<code>/a(b c)d/</code>	<code># abd or acd</code>
<code>/a(b c)+d/</code>	<code># a, one or more b's or c's, d</code>
<code>/a[bcd]e/</code>	<code># abe or ace or ade</code>
<code>/a[A-Za-z0-9]b/</code>	<code># a, letter or digit, b</code>
<code>/a[^A-Za-z]b/</code>	<code># a, any character but a letter, b</code>

# Character Class Shortcuts

- Perl provides shortcuts for commonly used character classes.

Digit char: `\d == [0-9]`

Word char: `\w == [A-Za-z0-9_]`

whiteSpace char: `\s == [ \f\t\n\r ]`

nonDigit: `\D == [^\d]`

nonWord: `\W == [^\w]`

non whiteSpace: `\S == [^\s]`

# General Quantifiers

- Can use {min,max} to represent the number of repetitions for an item in a regular expression.

a{1,3}      # a, aa, or aaa

a{5,5}      # aaaaa

a{5}        # aaaaa

a{2,}       # two or more a's

a{0,}       # a\*

a{1,}       # a+

a{0,1}      # a?

# Anchors

- Perl anchors provide context in which a pattern is matched.

`/^a/` # matches a if after the beginning of the line

`/a$/` # matches a if before the end of the line

`/^a$/` # matches a if it is a complete line

`/\ba/` # matches a if it is at the start of a word

`/a\b/` # matches a if it is at the end of a word

`/\ba\b/` # matches a if it is a complete word

# Remembering Substring Matches

- (...) is used for not only grouping, but also for remembering substrings in a pattern match. Note there are similar features in the sed Unix utility.
- Can refer to these substrings.
  - Backreferences can be used inside the pattern to refer to the memory saved earlier in the current pattern.
  - Memory variables can be used outside of the pattern to refer to the memory saved in the last pattern.

# Backreferences

- A backreference has the form  $\backslash number$ . It indicates the string matching the memory reference in the current pattern identified by that *number*. In numbering backreferences, you can just count the left parentheses.

`/(a|b)\1/` # match aa or bb

`/((a|b)c)\1/` # match acac or bcbc

`/((a|b)c)\2/` # match aca or bcb

`/(.)\1/` # match any character but newline that appears  
# twice in a row

`/(\w+)\s+\1/` # match any word that appears twice in a row and  
# is separated by one or more whitespace chars

`/(["'`]).*\1/` # match string enclosed by '...' or "..."

# Memory Variables

- A memory variable has the form  $\$number$ . It indicates the string in the last pattern matching the memory reference identified by that *number*.

# Checks if  $\$_$  has a word and prints that word.

```
if ( /\s+(\w+)\s+/ ) {  
    print $1, "\n";  
}
```

# If  $\$_$  has a '\$' followed by 1 to 3 digits and optionally  
# followed by groups of a comma with 3 digits, then print  
# the price.

```
if ( /(\$\d{1,3}(,\d{3})*)/ ) {  
    print "The price is $1.\n";  
}
```

# Binding Operator

- So far we have only seen checks for patterns in \$\_. We can check for patterns in arbitrary strings using the =~ and !~ match operators.
- General form:
  - # check if <pattern> match for <string>  
<string> =~ /<pattern>/
  - # check if there is not a <pattern> match for <string>  
<string> !~ /<pattern>/

# Example of Using Binding Operators

# If the user did not specify to exit, then print the line.

```
if ($line !~ /\bexit\b/) {  
    print $line;  
}
```

# If a blank line, then proceed to the next iteration.

```
if ($line =~ /^$/) {  
    next;  
}
```

# Automatic Match Variables

- A pattern only has to match a portion of a string to return a true value. There are some automatic match variables that do not require parentheses to be specified within the pattern.

`$`` # contains portion of the string before the match

`$&` # contains portion of the string that matched

`$'` # contains portion of the string after the match

# Automatic Match Variable Examples

```
# establish relationship
if ( $line =~ / is the parent of / ) {
    print “ $' is the child of $`\n” ;
}
```

```
# change the assignment operator
if ( $line =~ /=/ ) {
    print “ $`:= $” ;
}
```

```
# find the first word in the line
if ( $line =~ /\b\w+\b/ ) {
    print “ $& is the first word in the line.\n” ;
}
```

# Using Other Pattern Delimiters

- You can use other delimiters besides slashes for patterns, as we saw with the `qw` shortcut for quoted words in a list. If you do use a different delimiter, then you must precede the first delimiter with an 'm'. The 'm' is optional when using slashes. Note some delimiters are paired and others are nonpaired.

`m/.../`

`m{...}`

`m[...]`

`m(...)`

`m!...!`

`m,....,`

`m^...^`

`m#...#`

- You should probably use slashes unless your pattern contains slashes.

# Example of Using Other Pattern Delimiters

- The pattern matching can be more readable when using a pattern delimiter other than a '/' when the pattern contains a '/'.  
# Search for the start of a URL.  
if (\$s =~ /http:\^\/\^)

# Search for the start of a URL.

if (\$s =~ m^http://^)

# Search for the start of a URL.

if (\$s =~ m^http://^)

# Option Modifiers

- There are a set of letters that you can place after the last delimiter in a pattern to indicate how the pattern is to be interpreted.

<u>Modifier</u>	<u>Description</u>
i	case-insensitive matching
s	treat string as a single line
g	find all occurrences

# Case-Insensitive Matching

- You can make a case-insensitive pattern match by putting 'i' as an option modifier after the last delimiter.

```
\b[Uu][Nn][Ii][Xx]\b/ # matches the word unix  
                        # regardless of case  
\bunix\b/i             # same as above
```

# Matching Any Character

- The '.' character in a pattern indicates to match any character but a newline. By using the 's' option modifier, the '.' character will also match newlines.

# Matching a quoted string that could contain newlines.

```
/"(.|\n)*"/
```

# A more concise pattern.

```
/".*"/s
```

# Global Pattern Matching

- You can use the 'g' option modifier to find each match of a pattern in a string. Perl remembers the match position where it left off the last time it matched the string and will find the next match. If the string is a variable and it is modified in any way, then the match position is reset to the beginning of the string.

```
# print each acronym in a string on a separate line
while ($s =~ /[A-Z]{2,}/g) {
    print "$&\n";
}
```

# Interpolating Patterns

- The regular expressions allow interpolation just as double quoted strings. Thus, patterns could be read in at run time and used to match strings.

```
# match dynamic pattern if it occurs at beginning of a line
if ($line =~ m/^\$var/) {
    print $line;
}
```

- Note that the Perl program may fail if the regular expression comprising the pattern is invalid.

# Performing Substitutions

- The `s/.../.../` form can be used to make substitutions in the specified string. Note that if paired delimiters are used, then you have to use two pairs of the delimiters. 'g' after the last delimiter indicates to replace more than just the first occurrence. The substitution can be bound to a string using “=~”. Otherwise it makes the substitutions in `$_`. The operation returns the number of replacements performed, which can be more than one with the 'g' option.

```
# “search” is replaced with “replace” for all occurrences in $s  
# and the number of replacements is assigned to $var  
$var = $s =~ s/search/replace/g;
```

# Substitution Examples

```
s/\bfigure (\d+)/Figure $1/ # capitalize references to figures
s{//(.*)}{^*$1\*/} # use old style C comments
s!\bif(!if (! # put a blank between “if” and “(”
s(!)(.) # tone down that message
s[!][.]g # replace all occurrences of '!' with '.'
```

# Case Shifting

- In the replacement string, you can force what follows a given point in the replacement string to be upper or lower case by using the `\U` or `\L` indicators, respectively.

```
# change acm or ieee to uppercase within $text  
$text =~ s/(acm|ieee)\U$1/;
```

```
# change course prefix to lowercase in $text and assign  
# to $num the number of replacements made  
$num = $text =~ s/\b(COP|CDA)\d+\L$&/g;
```

# Performing Translations

- In Perl you can also convert one set of characters to another using the `tr/.../.../` form. However, rather than specifying a pattern, you specify two strings in a manner similar to the *tr* Unix utility. Any character found that is in the first string is replaced with the corresponding character in the second string. It returns the number of characters replaced or deleted. If the replacement string is empty, then the search string is used by default and there is no effect on the string being searched. If there are fewer replacement characters, then the final one is replicated. The *d* modifier deletes characters not given a replacement. The *s* modifier squashes duplicate replaced characters.

# Translation Examples

# convert letters in \$text to lowercase

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

# count the digits in \$\_ and assign to \$cnt

```
$cnt = tr/0-9//;
```

# get rid of redundant blanks in \$\_

```
tr//s/;
```

# delete \*'s in \$text

```
$text =~ tr/\*//d;
```

# replace [ and { with ( in \$text

```
$text =~ tr/[{/(/;
```



# Join Function

- The join function has the opposite effect of the split operator. It takes a list of strings and concatenates them together into a single string. The first argument is a glue string and the remaining arguments are a list and it returns a string containing the remaining arguments separated by the glue string.

```
@fields = ("This", "sentence", "contains", "five", "words.");
```

```
# The statement below has the following effect:
```

```
# $line = "This sentence contains five words.";
```

```
$line = join " ", @fields;
```

# Filehandles

- A filehandle is an I/O connection between your process and some device or file.
- Perl has three predefined filehandles.

STDIN – standard input

STDOUT – standard output

STDERR – standard error

# Opening Filehandles

- You can open your own filehandle. Unlike other variables, filehandles are not declared with the `my` operator. The convention is to use all uppercase letters when referring to a filehandle.
- The `open` operator takes two arguments, a filehandle name and a connection (e.g. filename). The connection can start with “<”, “>”, or “>>” to indicate read, write, and append access.

```
open IN, “in.dat” ;           # open “in.dat” for input
open IN2, “<$file” ;         # open filename in $file for input
open OUT, “>out.dat” ;       # open “out.dat” for output
open LOG, “>>log.txt” ;     # open “log.txt” to append output
```

# Closing Filehandles

- The close operator closes a filehandle. This causes any remaining output data associated with this filehandle to be flushed to the file. Perl automatically closes a filehandle if you reopen it or if you exit the program.

```
close IN;      # closes the IN filehandle
```

```
close OUT;    # closes the OUT filehandle
```

```
close LOG;    # closes the LOG filehandle
```

# Exiting the Process

- You can exit a process by using the `exit` function. It takes an argument that indicates the exit status.

```
exit 0;    # everything is fine
```

```
exit 1;    # something went wrong
```

- You can also exit a process by using the `die` function, which in addition prints a message to `STDERR`. Perl also automatically appends the name of the program and the current line to the message.

```
die "Something went wrong." ;
```

# Checking the Status of Open

- You can check the status of opening a file by examining the result of the open operation. It returns true for a successful open and false for failure.

```
if (!open OUT, ">out.dat" ) {  
    die "Could not open out.dat." ;  
}
```

# Using Filehandles

- After opening a filehandle, you can use it to read or write depending on how you opened it. Note that in a print or printf statement, the filehandle name is not followed by a comma.

```
Open IN, "<in.dat";
Open OUT, ">out.dat";
$i = 1;
while ($line = <IN>) {
    printf OUT "%d: $line", $i;
}
```

# Reopening a Standard Filename

- You can reopen a standard filename. This allows you to perform input or output in a normal fashion, but to redirect the I/O from/to a file within the Perl program.

```
# redirect standard output to "out.txt"  
open STDOUT, ">out.txt";  
printf "Hello world!\n";
```

```
# redirect standard error to append to "log.txt"  
open STDERR, ">>log.txt";
```

# Checking the Status of a File

- You can check the status of a file by performing a file test. Each file test returns a boolean value that can be referenced in control structures. These file tests are similar to those available in the shell.
- General form.
  - *option filename*
- Some common options are:
  - r (file is readable), -w (file is writeable), -x (file is executable),
  - e (file exists), -f (is a plain file), -d (is a directory)

# Example of Checking the Status of a File

```
# open the file
if (! open IN, $filename) {

    # print the reason why the file could not be opened
    if (! -e $filename) {
        die “$filename does not exist.”;
    }
    if (! -r $filename) {
        die “$filename is not readable.”;
    }
    if (-d $filename) {
        die “$filename should not be a directory.”;
    }
    die “$filename could not be opened.”;
}
```

# Defining Subroutines in Perl

- Perl also supports subroutines (i.e. functions). Declarations of subroutines can be placed anywhere.

- General form.

```
sub <name> {  
    <one_or_more_statements>  
}
```

- Example

```
sub read_fields {  
    $line = <>;  
    @fields = split //, $line;  
}
```

# Invoking Subroutines

- You invoke a subroutine by preceding the name with an '&' character.

```
&read_fields;
```

```
printf OUT “The number of fields is %d.\n”, $#fields+1;
```

# Return Values

- A Perl subroutine can return a value.
  - The result of the last calculation performed is the return value.
  - A value can be explicitly returned with the return statement.

```
sub maxsize {  
    1000;  
}
```

```
sub nextval {  
    $val++;  
    return $val;  
}
```

# Subroutine Arguments

- To pass arguments to a subroutine, simply put a list expression in parentheses after the name of the subroutine when you invoke it.
- The arguments are received in the array `@_`. You can determine the number of arguments by examining `$_#`. You can access individual arguments using `$_[0]`, `$_[1]`, etc. So this feature allows routines to be easily written to handle a variable number of arguments.

# Example Use of Arguments

```
sub max {  
  my $maxnum = shift @_; # shift the first argument off  
  foreach $val (@_) { # for each remaining argument  
    if ($val > $maxnum) {  
      $maxnum = $val;  
    }  
  }  
  return $maxnum;  
}
```

```
$num = &max($a, $b, $c); # could pass one or more args
```

# Variables within Subroutines

- Variables declared with the my operator are only visible within that block. Thus, variables declared within a subroutine are only visible within that subroutine. This means that the declaration of a variable within a block will prevent access to a variable with the same name outside the block.

# Sort Subroutines

- The sort operator can accept the name of a subroutine to determine the order between a pair of elements. Rather than receiving arguments in `@_`, the sort subroutine instead receives arguments in `$a` and `$b`. It returns a `-1` if `$a` should appear before `$b`, a `1` if `$b` should appear before `$a`, and `0` if the order does not matter.

# Sorting Operators

- The `<=>` operator returns -1, 0, or 1 depending on the numerical relationship between the two operands.

```
sub by_number {  
    return $a <=> $b;  
}
```

- The `cmp` operator returns -1, 0, or 1 depending on the string comparison relationship between the two operands. By default the sort operator does this type of comparison.

```
sub by_ASCII {  
    return $a cmp $b;  
}
```

# Using Sorting Functions

@vals = sort by\_number @vals; # sort @vals numerically

@s = sort by\_ASCII @fields; # sort @fields in ASCII order

@vals = sort {\$a <=> \$b} @vals; # numerical sort specified  
# inline

@s = sort {\$a cmp \$b} @fields; # ASCII sort specified inline

@k = sort {\$n{\$a} cmp \$n{\$b}} keys %n;  
# returns list of keys based on  
ASCII sort of hash values

# Picking Items from a List with Grep

- The `grep` operator extracts items from a list. The first argument is a function that returns true or false. The remaining arguments are the list of items. The `grep` operator returns a list. The function uses `$_` to access each item in the list. This is really a shortcut to avoid using a `foreach` statement.

```
foreach @vals {                               # extract the odd values
    if ($_ % 2) {
        push @oddvals, $_;
    }
}
```

```
# extract the odd values in a single line of code
@oddvals = grep { $_ % 2 } @vals;
```

# Transforming Items with a Map

- The map operator is similar to grep, except that the value returned from the function is always added to the resulting list returned by map.

```
# take the absolute values of each element
```

```
foreach (@vals) {  
    if ($_ < 0)  
        push @absvals, -$_;  
    else  
        push @absvals, $_;  
}
```

```
# take the absolute values in a single line of code
```

```
@absvals = map { $_ < 0 ? -$_ : $_ } @vals;
```

# Directory Operations

- Perl provides directory operations that are portable across different operating systems. The general form and an example of each given below.

<code>chdir <i>dirname</i>;</code>	<code># cd to a new directory</code>
<code>chdir “asg1”;</code>	<code># cd to subdirectory “asg1”</code>
<code>glob <i>filename_pattern</i>;</code>	<code># return a list of filenames</code>
<code>glob “*.c”</code>	<code># return list of *.c filenames</code>

# Manipulating Files and Directories

- Perl functions exist to change files and directories. The general form and an example of each are given below.

<code>unlink <i>filenames</i>;</code>	<code># <i>remove list of files</i></code>
<code>unlink in.dat, out.dat;</code>	<code># remove two files</code>
<code>rename <i>oldfile, newfile</i>;</code>	<code># <i>rename a file</i></code>
<code>rename “tmp.out”, “data.out”;</code>	<code># renamed an output file</code>
<code>mkdir <i>dirname, permissions</i>;</code>	<code># <i>make a new directory</i></code>
<code>mkdir “asg1”, 0700;</code>	<code># mkdir asg1 where only # the user can access it</code>
<code>rmdir <i>dirnames</i>;</code>	<code># <i>remove list of directories</i></code>
<code>rmdir “asg1”;</code>	<code># remove asg1 directory</code>

# Manipulating Files and Directories (cont.)

`chmod perms, filenames;`

`# change permissions`

`chmod 0755, "asg1"`

`# change permissions on asg1`

# Invoking Processes

- Can use the `system` command to create a child process.

```
system "date";      # invokes the Unix date command
```

- Can use backquotes to capture output.

```
$time = `date`;    # capture Unix date command output
```