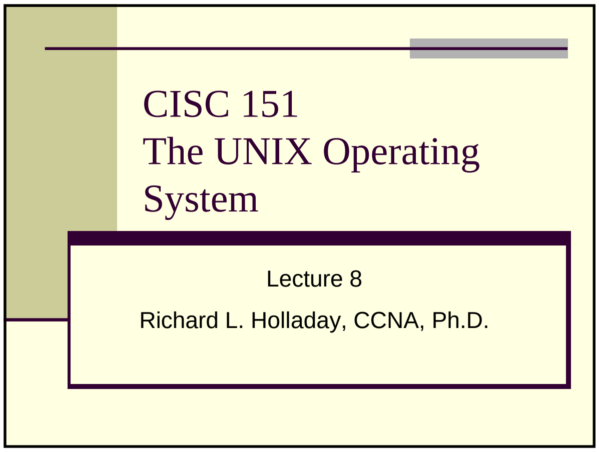
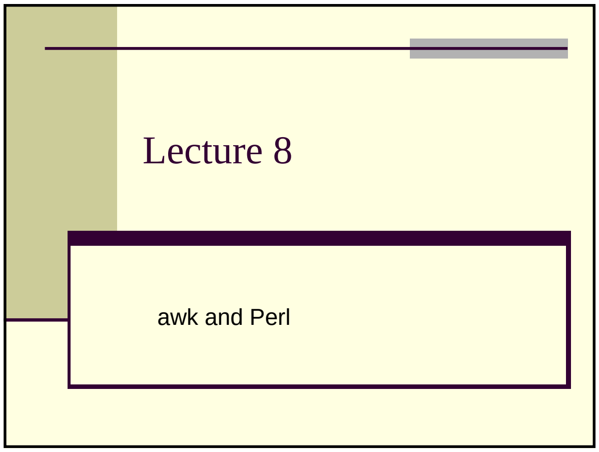


CISC 151
The UNIX Operating System

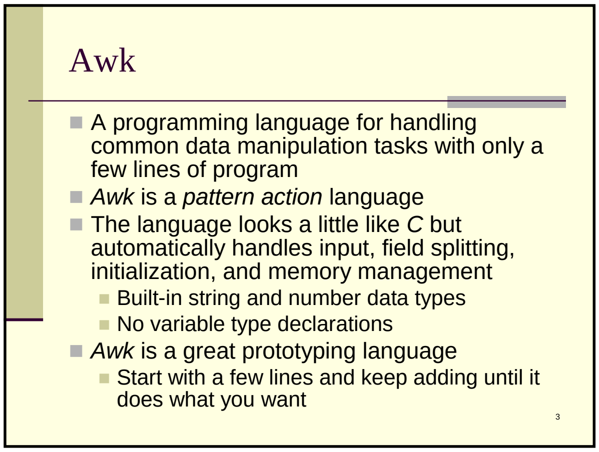
Lecture 8

Richard L. Holladay, CCNA, Ph.D.



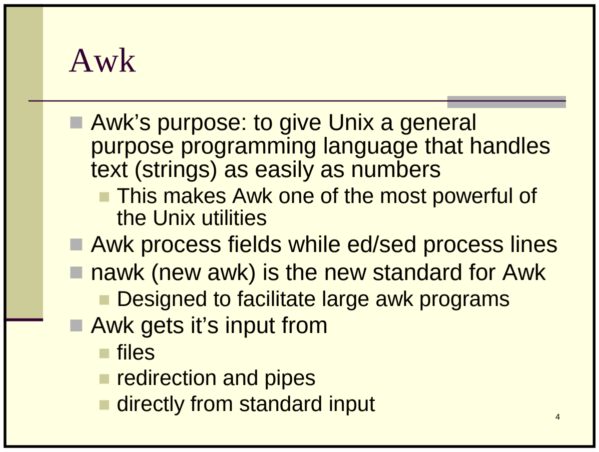
Lecture 8

awk and Perl



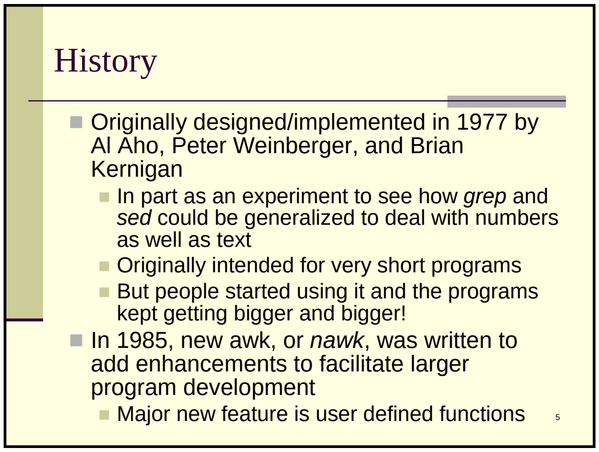
Awk

- A programming language for handling common data manipulation tasks with only a few lines of program
- *Awk* is a *pattern action* language
- The language looks a little like *C* but automatically handles input, field splitting, initialization, and memory management
  - Built-in string and number data types
  - No variable type declarations
- *Awk* is a great prototyping language
  - Start with a few lines and keep adding until it does what you want



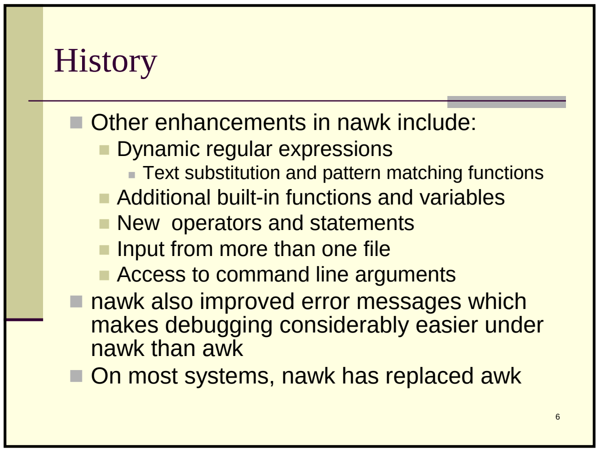
Awk

- Awk's purpose: to give Unix a general purpose programming language that handles text (strings) as easily as numbers
  - This makes Awk one of the most powerful of the Unix utilities
- Awk process fields while ed/sed process lines
- nawk (new awk) is the new standard for Awk
  - Designed to facilitate large awk programs
- Awk gets it's input from
  - files
  - redirection and pipes
  - directly from standard input



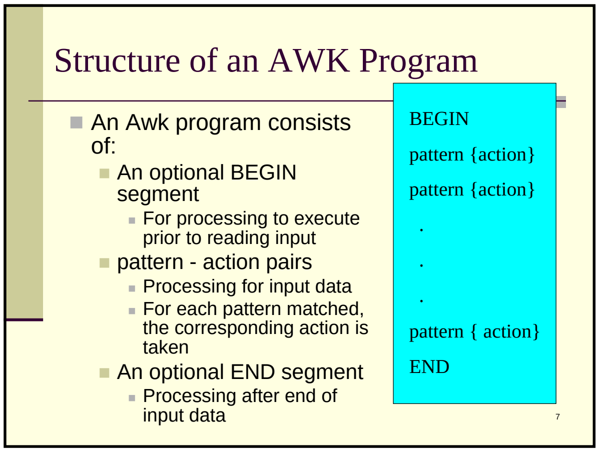
History

- Originally designed/implemented in 1977 by Al Aho, Peter Weinberger, and Brian Kernigan
  - In part as an experiment to see how *grep* and *sed* could be generalized to deal with numbers as well as text
  - Originally intended for very short programs
  - But people started using it and the programs kept getting bigger and bigger!
- In 1985, new awk, or *nawk*, was written to add enhancements to facilitate larger program development
  - Major new feature is user defined functions



History

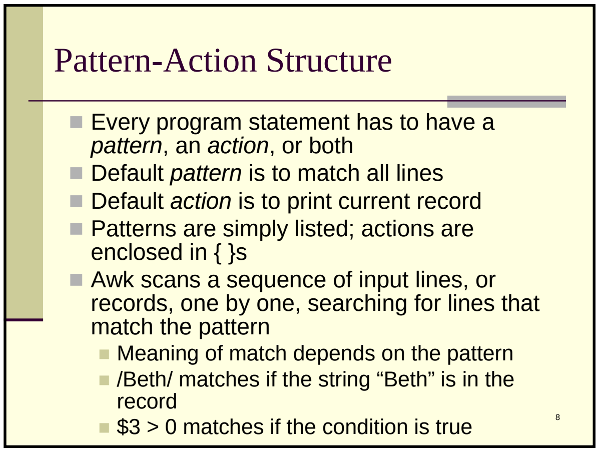
- Other enhancements in nawk include:
  - Dynamic regular expressions
    - Text substitution and pattern matching functions
  - Additional built-in functions and variables
  - New operators and statements
  - Input from more than one file
  - Access to command line arguments
- nawk also improved error messages which makes debugging considerably easier under nawk than awk
- On most systems, nawk has replaced awk



Structure of an AWK Program

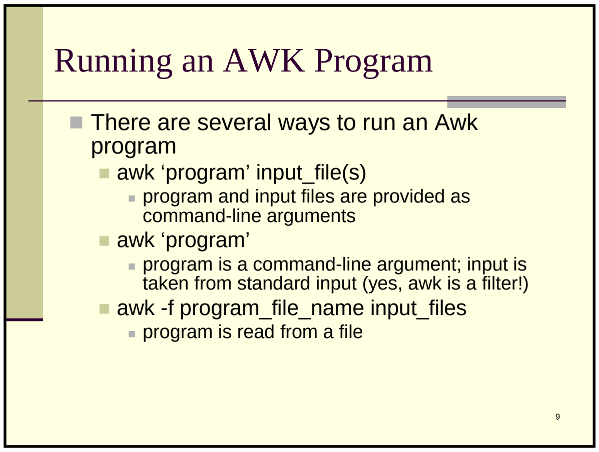
- An Awk program consists of:
  - An optional BEGIN segment
    - For processing to execute prior to reading input
  - pattern - action pairs
    - Processing for input data
    - For each pattern matched, the corresponding action is taken
  - An optional END segment
    - Processing after end of input data

```
BEGIN
pattern {action}
pattern {action}
  .
  .
  .
pattern { action}
END
```



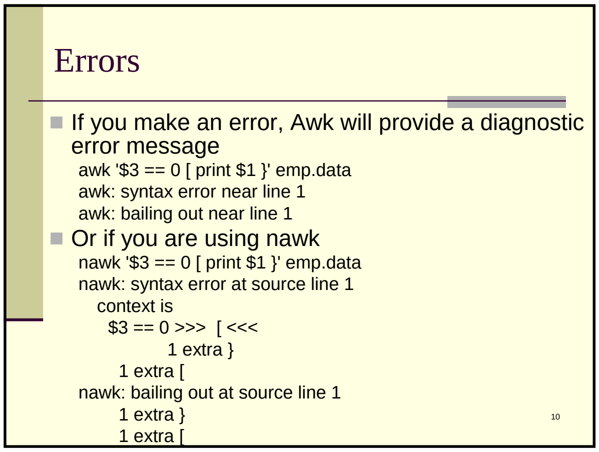
Pattern-Action Structure

- Every program statement has to have a *pattern*, an *action*, or both
- Default *pattern* is to match all lines
- Default *action* is to print current record
- Patterns are simply listed; actions are enclosed in { }s
- Awk scans a sequence of input lines, or records, one by one, searching for lines that match the pattern
  - Meaning of match depends on the pattern
  - /Beth/ matches if the string “Beth” is in the record
  - $3 > 0 matches if the condition is true



Running an AWK Program

- There are several ways to run an Awk program
  - awk ‘program’ input_file(s)
    - program and input files are provided as command-line arguments
  - awk ‘program’
    - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)
  - awk -f program_file_name input_files
    - program is read from a file



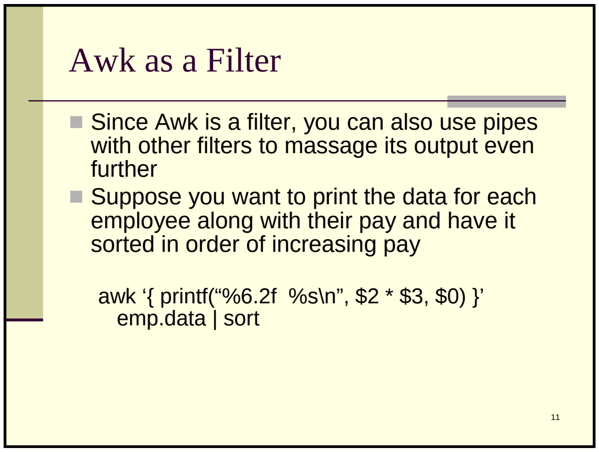
Errors

- If you make an error, Awk will provide a diagnostic error message

```
awk '$3 == 0 [ print $1 }' emp.data
awk: syntax error near line 1
awk: bailing out near line 1
```

- Or if you are using nawk

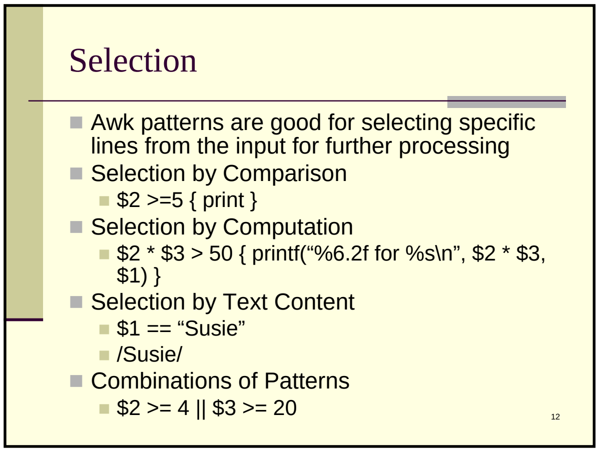
```
nawk '$3 == 0 [ print $1 }' emp.data
nawk: syntax error at source line 1
   context is
     $3 == 0 >>>  [ <<<
              1 extra }
       1 extra [
nawk: bailing out at source line 1
       1 extra }
       1 extra [
```



Awk as a Filter

- Since Awk is a filter, you can also use pipes with other filters to massage its output even further
- Suppose you want to print the data for each employee along with their pay and have it sorted in order of increasing pay

```
awk '{ printf("%6.2f  %s\n", $2 * $3, $0) }'
   emp.data | sort
```



Selection

- Awk patterns are good for selecting specific lines from the input for further processing
- Selection by Comparison
  - $2 >=5 { print }
- Selection by Computation
  - $2 * $3 > 50 { printf("%6.2f for %s\n", $2 * $3, $1) }
- Selection by Text Content
  - $1 == "Susie"
  - /Susie/
- Combinations of Patterns
  - $2 >= 4 || $3 >= 20

Data Validation

- Validating data is a common operation
- Awk is excellent at data validation
  - NF != 3 { print $0, “number of fields not equal to 3” }
  - $2 < 3.35 { print $0, “rate is below minimum wage” }
  - $2 > 10 { print $0, “rate exceeds $10 per hour” }
  - $3 < 0 { print $0, “negative hours worked” }
  - $3 > 60 { print $0, “too many hours worked” }

BEGIN and END

- Special pattern BEGIN matches before the first input line is read; END matches after the last input line has been read
- This allows for initial and wrap-up processing

```
BEGIN { print “NAME    RATE    HOURS”; print “” }
        { print }
END    { print “total number of employees is”, NR }
```

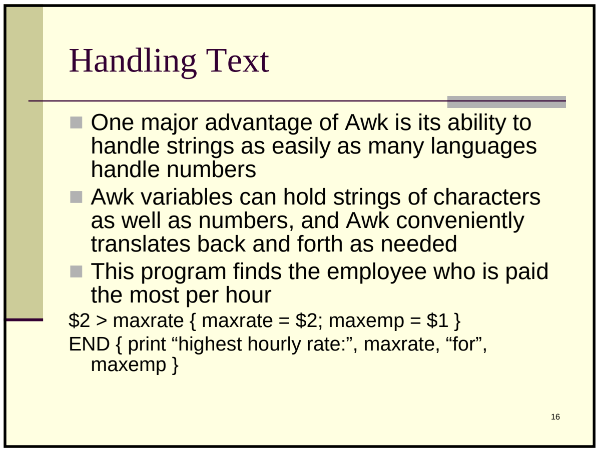
Computing with AWK

- Counting is easy to do with Awk

```
$3 > 15 { emp = emp + 1}
END    { print emp, “employees worked more than 15 hrs”}
```

- Computing Sums and Averages is also simple

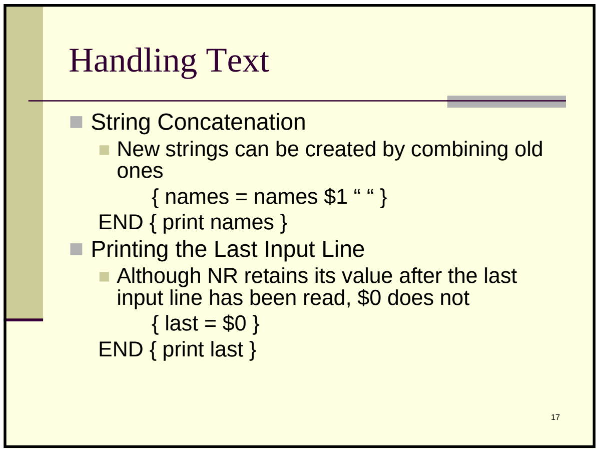
```
        { pay = pay + $2 * $3 }
  END { print NR, “employees”
          print “total pay is”, pay
          print “average pay is”, pay/NR
        }
```



Handling Text

- One major advantage of Awk is its ability to handle strings as easily as many languages handle numbers
- Awk variables can hold strings of characters as well as numbers, and Awk conveniently translates back and forth as needed
- This program finds the employee who is paid the most per hour

```
$2 > maxrate { maxrate = $2; maxemp = $1 }
END { print “highest hourly rate:”, maxrate, “for”,
    maxemp }
```



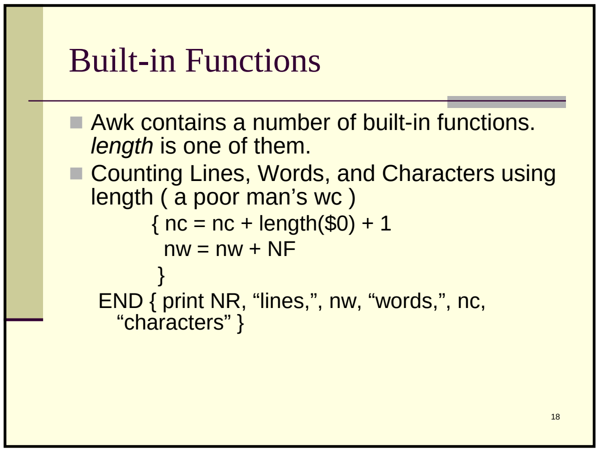
Handling Text

- String Concatenation
  - New strings can be created by combining old ones

```
        { names = names $1 “ “ }
 END { print names }
```

- Printing the Last Input Line
  - Although NR retains its value after the last input line has been read, $0 does not

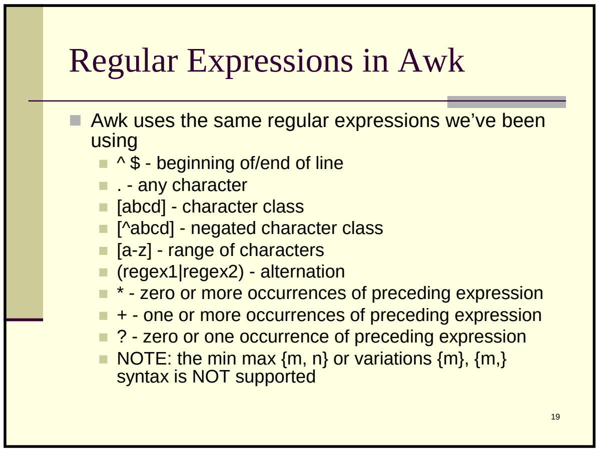
```
        { last = $0 }
 END { print last }
```



Built-in Functions

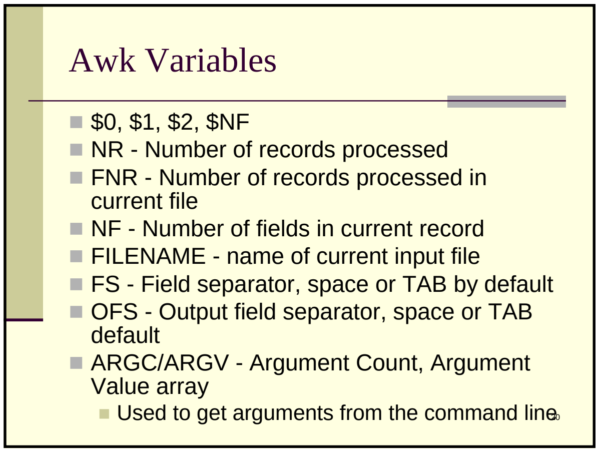
- Awk contains a number of built-in functions. *length* is one of them.
- Counting Lines, Words, and Characters using length (a poor man’s wc)

```
        { nc = nc + length($0) + 1
          nw = nw + NF
         }
 END { print NR, “lines,”, nw, “words,”, nc,
    “characters” }
```



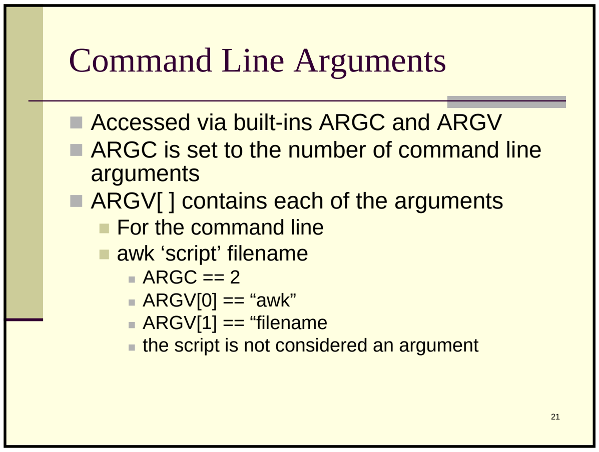
Regular Expressions in Awk

- Awk uses the same regular expressions we’ve been using
  - ^ $ - beginning of/end of line
  - . - any character
  - [abcd] - character class
  - [^abcd] - negated character class
  - [a-z] - range of characters
  - (regex1|regex2) - alternation
  - * - zero or more occurrences of preceding expression
  - + - one or more occurrences of preceding expression
  - ? - zero or one occurrence of preceding expression
  - NOTE: the min max {m, n} or variations {m}, {m,} syntax is NOT supported



Awk Variables

- $0, $1, $2, $NF
- NR - Number of records processed
- FNR - Number of records processed in current file
- NF - Number of fields in current record
- FILENAME - name of current input file
- FS - Field separator, space or TAB by default
- OFS - Output field separator, space or TAB default
- ARGC/ARGV - Argument Count, Argument Value array
  - Used to get arguments from the command line



Command Line Arguments

- Accessed via built-ins ARGC and ARGV
- ARGC is set to the number of command line arguments
- ARGV[ ] contains each of the arguments
  - For the command line
  - awk ‘script’ filename
    - ARGC == 2
    - ARGV[0] == “awk”
    - ARGV[1] == “filename
    - the script is not considered an argument

Command Line Arguments

- ARGC and ARGV can be used like any other variable
- The can be assigned, compared, used in expressions, printed
- They are commonly used for verifying that the correct number of arguments were provided

Operators

- = assignment operator; sets a variable equal to a value or string
- == equality operator; returns TRUE is both sides are equal
- != inverse equality operator
- && logical AND
- || logical OR
- ! logical NOT
- <, >, <=, >= relational operators
- +, -, /, *, %, ^
- String concatenation

Control Flow Statements

- Awk provides several control flow statements for making decisions and writing loops
- If-Else

```
if (expression is true or non-zero){
        statement1
        }
else {
        statement2
        }
```

where *statement1* and/or *statement2* can be multiple statements enclosed in curly braces { }s

- the *else* and associated *statement2* are optional

Loop Control

- While

```
while (expression is true or non-zero) {
            statement1
            }
```

For Loops

- For

```
for(expression1; expression2; expression3) {
    statement1
    }
```

  - This has the same effect as:

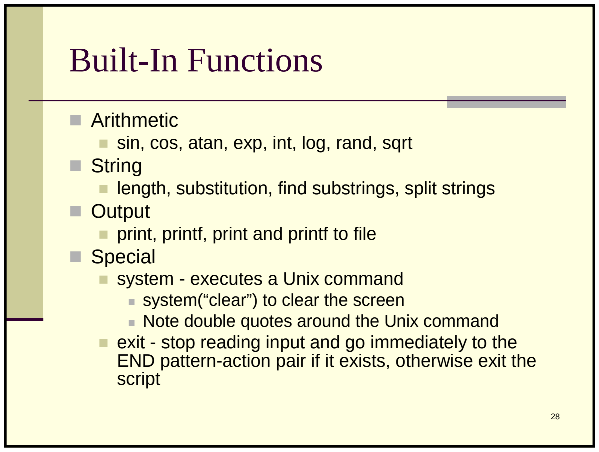
```
expression1
while (expression2) {
  statement1
  expression3
  }
```

  - for(;;) is an infinite loop

Do While

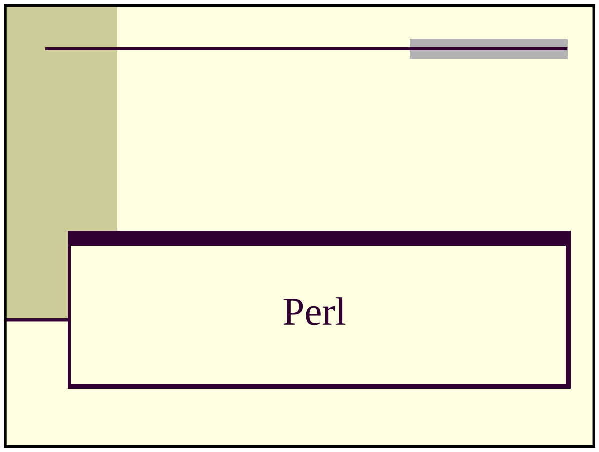
- Do While

```
do {
    statement1
    }
while (expression)
```

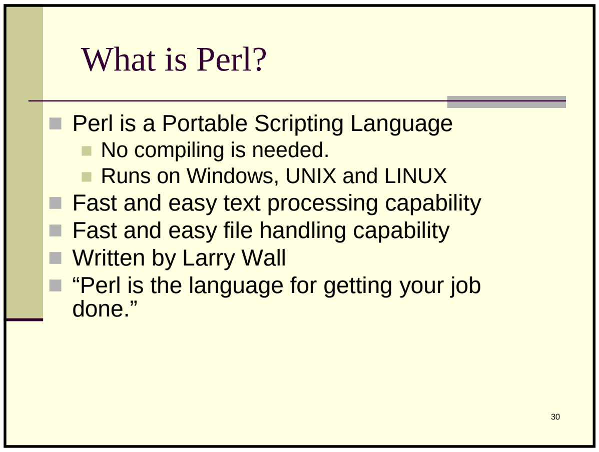


Built-In Functions

- Arithmetic
  - sin, cos, atan, exp, int, log, rand, sqrt
- String
  - length, substitution, find substrings, split strings
- Output
  - print, printf, print and printf to file
- Special
  - system - executes a Unix command
    - system(“clear”) to clear the screen
    - Note double quotes around the Unix command
  - exit - stop reading input and go immediately to the END pattern-action pair if it exists, otherwise exit the script



Perl



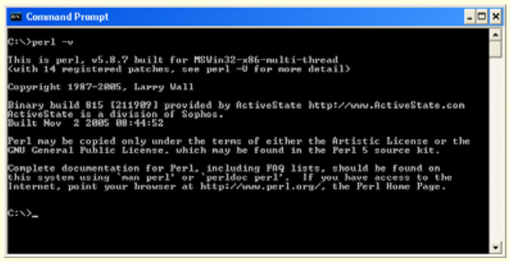
What is Perl?

- Perl is a Portable Scripting Language
  - No compiling is needed.
  - Runs on Windows, UNIX and LINUX
- Fast and easy text processing capability
- Fast and easy file handling capability
- Written by Larry Wall
- “Perl is the language for getting your job done.”

How to Access Perl

- From Mesa’s network
- To install at home
  - www.perl.com Has rpm's for Linux
  - www.activestate.com Has free binaries for Windows
    - Other languages: Python, Tcl, and others
- Latest Stable Version is 5.8.7
- Development Release 5.9.2
  - To check if Perl is working and the version number
    - % perl –v

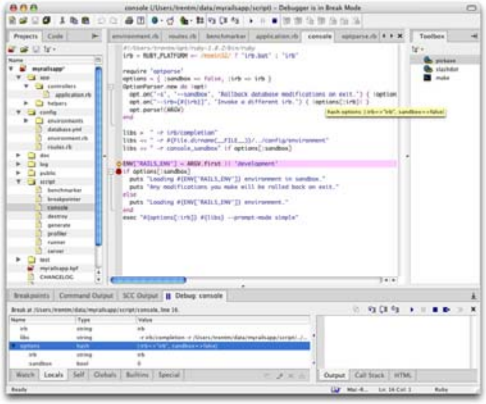
Perl under Windows



```
C:\>perl -v

This is perl, v5.8.7 built for MSWin32-x86-multi-thread
(with 14 registered patches, see perl -V for more detail)

Copyright 1987-2005, Larry Wall

Binary build 815 [211909] provided by ActiveState http://www.ActiveState.com
ActiveState is a division of Sophos.
Built Nov  2 2005 08:44:52

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using `man perl' or `perldoc perl'.  If you have access to the
Internet, point your browser at http://www.perl.org/, the Perl Home Page.


C:\>_
```

Komodo

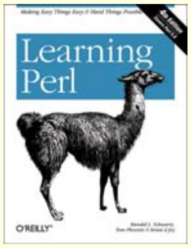
- Komodo is ActiveState's cross-platform, multi-language Integrated Development Environment (IDE)
  - Similar to Microsoft’s Visual Studio IDE
- Komodo supports development in numerous UNIX-based languages, including Perl, Python, PHP, XSLT, Tcl, JavaScript, and more.
- Excellent Tutorials, especially for Perl
- Runs on Linux, Mac OS X, Solaris, and Windows.
- ActiveState has free Windows versions of Perl, Python, and Tcl that can be downloaded and run seamlessly in Komodo (or from the command-line)
- Cost for Komodo:
  - Work and Commercial Use: $295
  - Personal & Student Use: $29.95



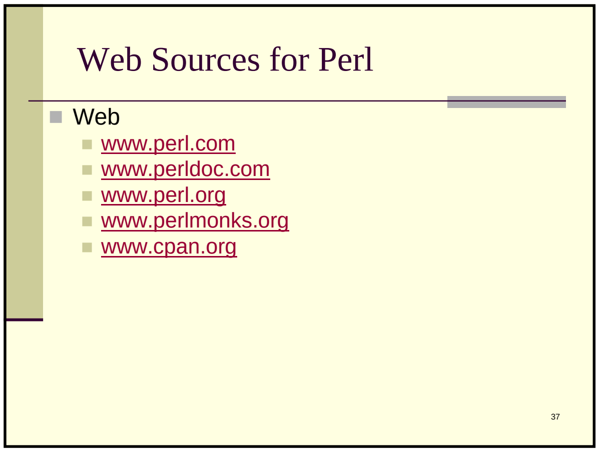
Resources For Perl

- Books:
  - *Programming Perl* (The “Camel”)
    - By Larry Wall,Tom Christiansen and Jon Orwant
    - Published by O'Reilly
    - The Perl Bible
  - *Learning Perl* (The “Llama”)
    - By Randal L. Schwartz, Tom Phoenix, Brian D Foy
    - Published by O'Reilly
- Web Site
  - http://safari.oreilly.com
    - Contains both Programming Perl and Learning Perl in ebook form

The “Camel”

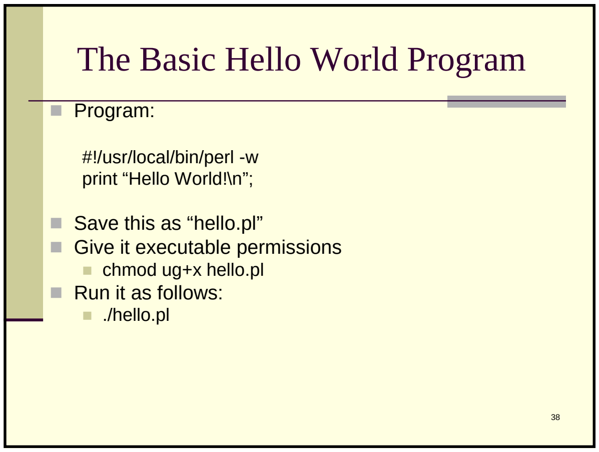


The “Llama”



Web Sources for Perl

- Web
  - www.perl.com
  - www.perldoc.com
  - www.perl.org
  - www.perlmonks.org
  - www.cpan.org

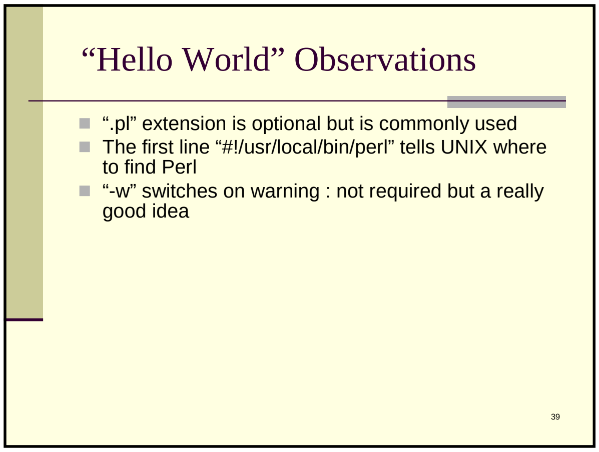


The Basic Hello World Program

- Program:

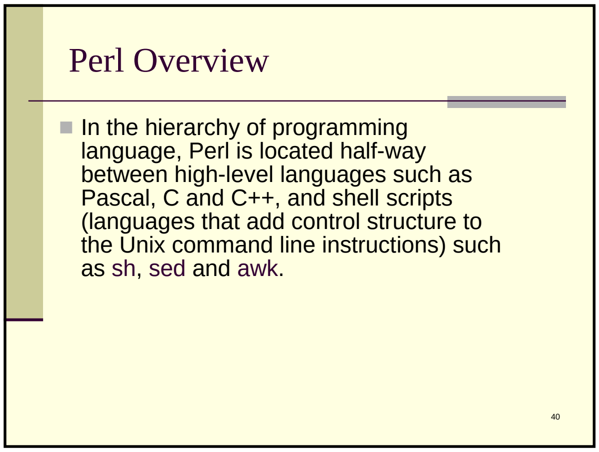
```
#!/usr/local/bin/perl -w
print “Hello World!\n”;
```

- Save this as “hello.pl”
- Give it executable permissions
  - chmod ug+x hello.pl
- Run it as follows:
  - ./hello.pl



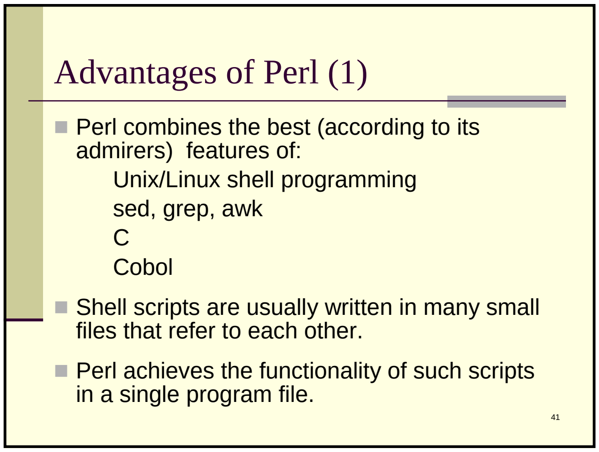
“Hello World” Observations

- “.pl” extension is optional but is commonly used
- The first line “#!/usr/local/bin/perl” tells UNIX where to find Perl
- “-w” switches on warning : not required but a really good idea



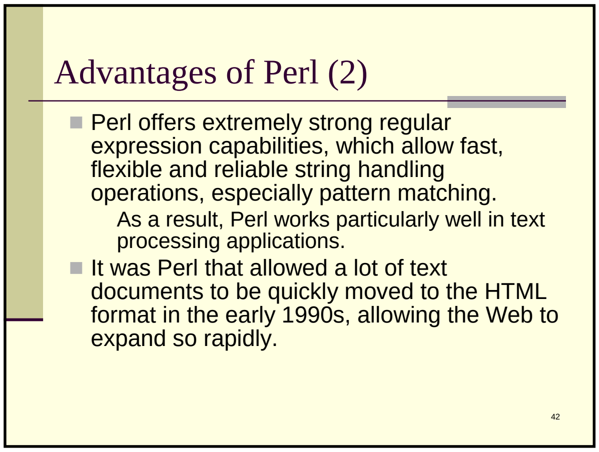
Perl Overview

- In the hierarchy of programming language, Perl is located half-way between high-level languages such as Pascal, C and C++, and shell scripts (languages that add control structure to the Unix command line instructions) such as sh, sed and awk.



Advantages of Perl (1)

- Perl combines the best (according to its admirers) features of:
  - Unix/Linux shell programming
  - sed, grep, awk
  - C
  - Cobol
- Shell scripts are usually written in many small files that refer to each other.
- Perl achieves the functionality of such scripts in a single program file.



Advantages of Perl (2)

- Perl offers extremely strong regular expression capabilities, which allow fast, flexible and reliable string handling operations, especially pattern matching.
  - As a result, Perl works particularly well in text processing applications.
- It was Perl that allowed a lot of text documents to be quickly moved to the HTML format in the early 1990s, allowing the Web to expand so rapidly.

Disadvantages of Perl

- Perl is a jumble! It contains many, many features from many languages and tools.
- It contains different constructs for the same functionality (for example, there are at least 5 ways to perform a one-line if statement).

  ➔It is not a very readable language.
- You cannot distribute a Perl program as an opaque binary. That is, you cannot really commercialize products you develop in Perl.

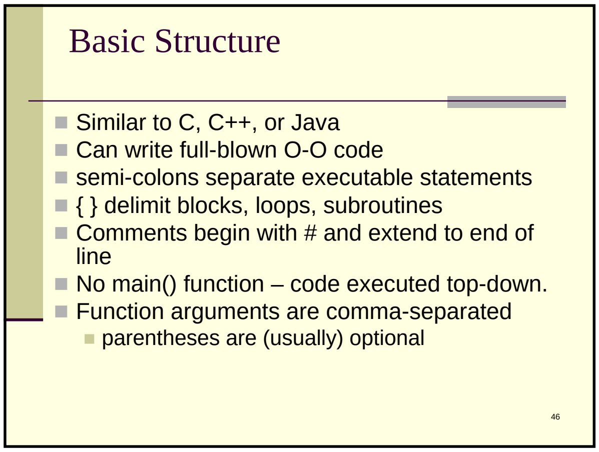
Perl resources and versions

- http://www.perl.org tells you everything that you want to know about Perl.
- Perl 1.0 released in 1987
- What you will see here is Perl 5.
- Current release: 5.8.7 released in July, 2002.
- Perl 6 (http://dev.perl.org/perl6/) is the next version, still under development, but moving along nicely. The first book on Perl 6 is in stores (http://www.oreilly.com/catalog/perl6es).

Perl is cited in the OED (Oxford English Dictionary

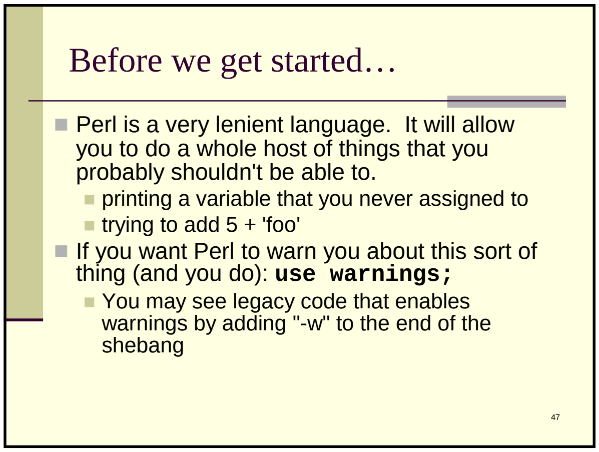
- Perl, *n.*

  PERL. [Alteration of pearl *n.*, with omission of *-a-* to differentiate it from an existing programming language called *PEARL*.
  Coined by Larry Wall in early 1987; the program was publicly released on 18 December of that year. Explanations of the name as an acronym (e.g. < the initial letters of *P*ractical *E*xtraction and *R*eport *L*anguage, < the initial letters of *P*athologically *E*clectic *R*ubbish *L*ister), although found early in the documentation for the language, are subsequent rationalizations.]
- A high-level interpreted programming language widely used for a variety of tasks and especially for applications running on the World Wide Web.
  The form *Perl* is preferred for the language itself; *perl* is used for the interpreter for the Perl language.



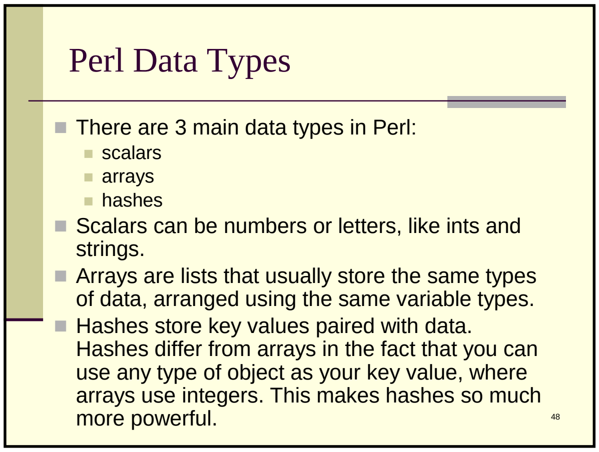
Basic Structure

- Similar to C, C++, or Java
- Can write full-blown O-O code
- semi-colons separate executable statements
- { } delimit blocks, loops, subroutines
- Comments begin with # and extend to end of line
- No main() function – code executed top-down.
- Function arguments are comma-separated
  - parentheses are (usually) optional



Before we get started…

- Perl is a very lenient language. It will allow you to do a whole host of things that you probably shouldn't be able to.
  - printing a variable that you never assigned to
  - trying to add 5 + 'foo'
- If you want Perl to warn you about this sort of thing (and you do): **`use warnings;`**
  - You may see legacy code that enables warnings by adding "-w" to the end of the shebang



Perl Data Types

- There are 3 main data types in Perl:
  - scalars
  - arrays
  - hashes
- Scalars can be numbers or letters, like ints and strings.
- Arrays are lists that usually store the same types of data, arranged using the same variable types.
- Hashes store key values paired with data. Hashes differ from arrays in the fact that you can use any type of object as your key value, where arrays use integers. This makes hashes so much more powerful.

Scalars

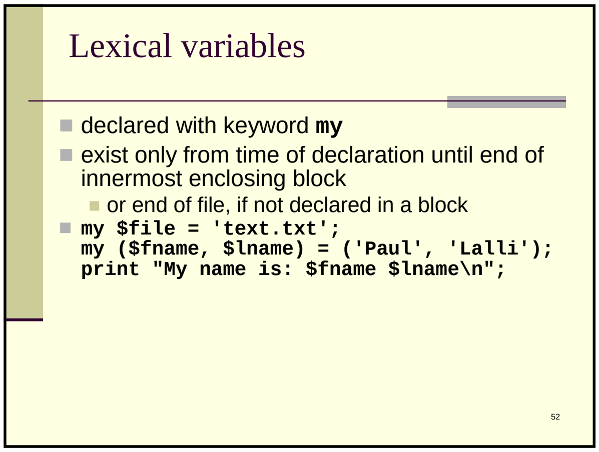
- A Scalar variable contains a single value.
- All of the standard types from C can be stored in a scalar variable
  - int, float, char, double, etc
  - No declaration of type of variable
- Scalar variable name starts with a $
- Next character is a letter or underscore
- Remaining characters: letters, numbers, or underscores.
  - name can be up to 255 characters long
    - don't do that.
- All scalars have a default value of **`undef`** before assigned a value. (Not to be confused with 5-character string 'undef')

Examples

- **`$num = 42;`**
- **`$letter = "a";`**
- **`$name = "Paul";`**
  - NOTE! Strings are *single values*!
- **`$grade = 99.44;`**
- **`$Big_String = "The Quick Brown Fox…";`**

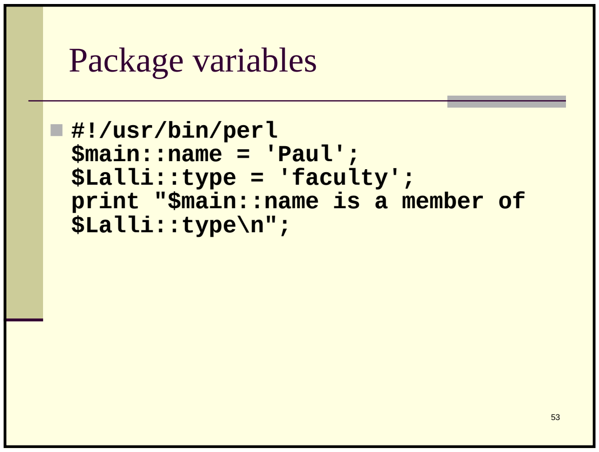
Package vs Lexical variables

- Package variables are global.
- Lexical variables are 'local' to innermost inclosing block/file
  - In Perl, 'local' means something else entirely
- Package variables belong to a given package, but can be accessed anywhere, by any piece of code.
  - default package is "main".
  - other packages declared with the **`package`** statement.
- Package variables are not declared. They simply exist.



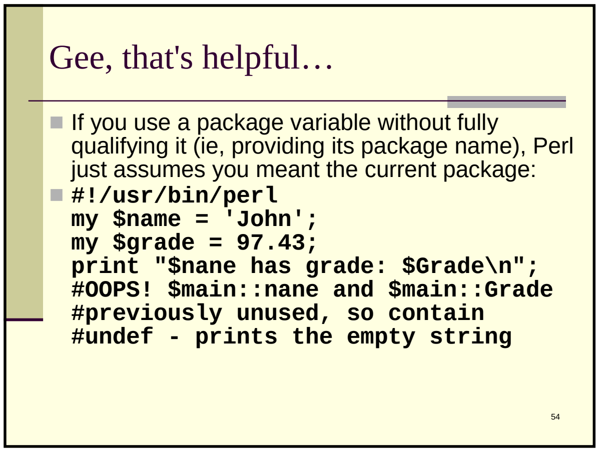
Lexical variables

- declared with keyword **`my`**
- exist only from time of declaration until end of innermost enclosing block
  - or end of file, if not declared in a block
-
```
my $file = 'text.txt';
my ($fname, $lname) = ('Paul', 'Lalli');
print "My name is: $fname $lname\n";
```



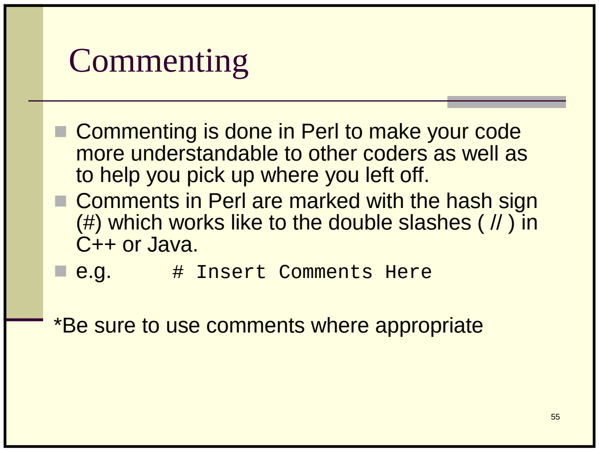
Package variables

-
```
#!/usr/bin/perl
$main::name = 'Paul';
$Lalli::type = 'faculty';
print "$main::name is a member of
$Lalli::type\n";
```



Gee, that's helpful…

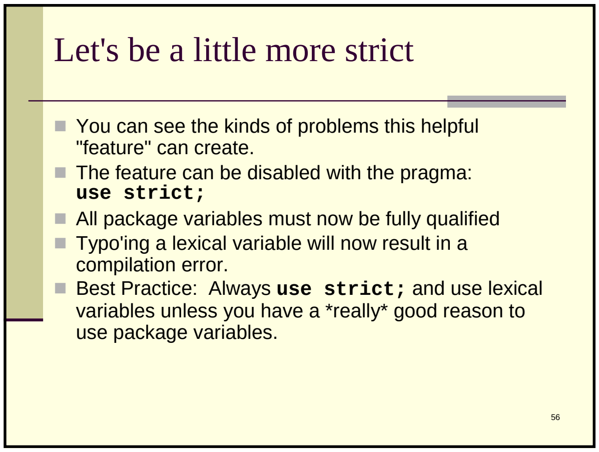
- If you use a package variable without fully qualifying it (ie, providing its package name), Perl just assumes you meant the current package:
-
```
#!/usr/bin/perl
my $name = 'John';
my $grade = 97.43;
print "$nane has grade: $Grade\n";
#OOPS! $main::nane and $main::Grade
#previously unused, so contain
#undef - prints the empty string
```



Commenting

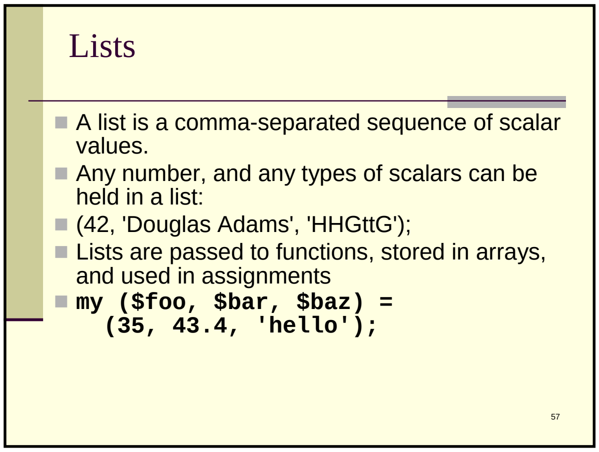
- Commenting is done in Perl to make your code more understandable to other coders as well as to help you pick up where you left off.
- Comments in Perl are marked with the hash sign (#) which works like to the double slashes (//) in C++ or Java.
- e.g. `# Insert Comments Here`

*Be sure to use comments where appropriate



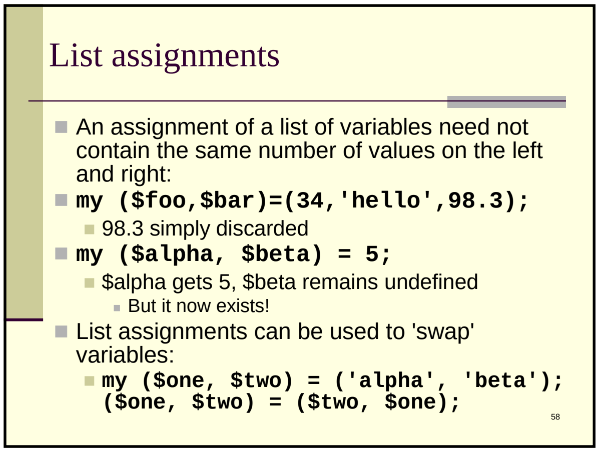
Let's be a little more strict

- You can see the kinds of problems this helpful "feature" can create.
- The feature can be disabled with the pragma: **`use strict;`**
- All package variables must now be fully qualified
- Typo'ing a lexical variable will now result in a compilation error.
- Best Practice: Always **`use strict;`** and use lexical variables unless you have a *really* good reason to use package variables.



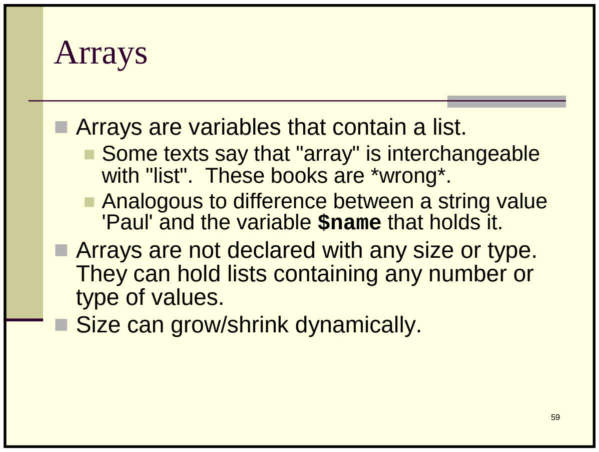
Lists

- A list is a comma-separated sequence of scalar values.
- Any number, and any types of scalars can be held in a list:
- (42, 'Douglas Adams', 'HHGttG');
- Lists are passed to functions, stored in arrays, and used in assignments
-
```
my ($foo, $bar, $baz) =
  (35, 43.4, 'hello');
```



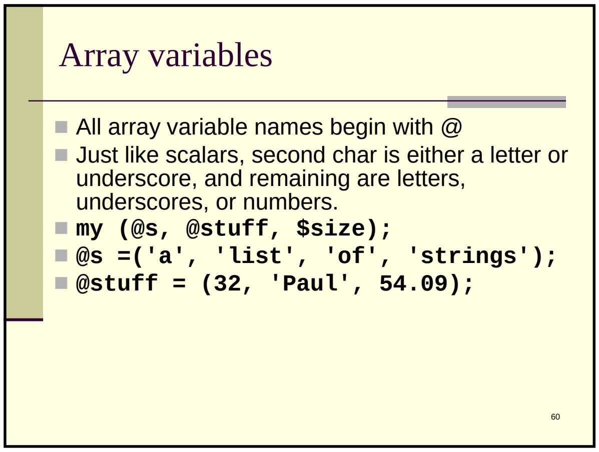
List assignments

- An assignment of a list of variables need not contain the same number of values on the left and right:
- **`my ($foo,$bar)=(34,'hello',98.3);`**
  - 98.3 simply discarded
- **`my ($alpha, $beta) = 5;`**
  - $alpha gets 5, $beta remains undefined
    - But it now exists!
- List assignments can be used to 'swap' variables:
  - **`my ($one, $two) = ('alpha', 'beta');`**
    `($one, $two) = ($two, $one);`



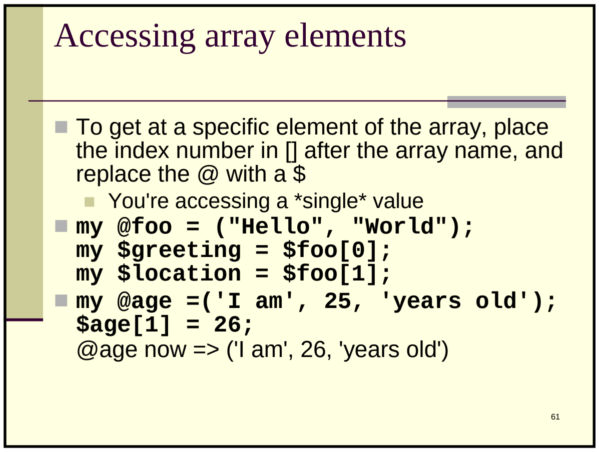
Arrays

- Arrays are variables that contain a list.
  - Some texts say that "array" is interchangeable with "list". These books are *wrong*.
  - Analogous to difference between a string value 'Paul' and the variable **`$name`** that holds it.
- Arrays are not declared with any size or type. They can hold lists containing any number or type of values.
- Size can grow/shrink dynamically.



Array variables

- All array variable names begin with @
- Just like scalars, second char is either a letter or underscore, and remaining are letters, underscores, or numbers.
- **`my (@s, @stuff, $size);`**
- **`@s =('a', 'list', 'of', 'strings');`**
- **`@stuff = (32, 'Paul', 54.09);`**

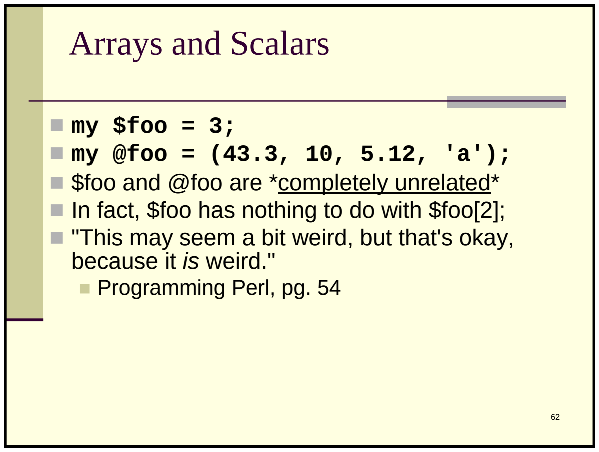


Accessing array elements

- To get at a specific element of the array, place the index number in [] after the array name, and replace the @ with a $
    - You're accessing a *single* value

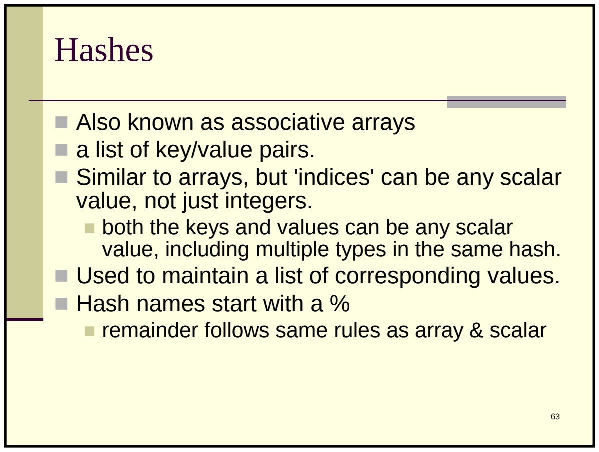
-
  ```
  my @foo = ("Hello", "World");
  my $greeting = $foo[0];
  my $location = $foo[1];
  ```

-
  ```
  my @age =('I am', 25, 'years old');
  $age[1] = 26;
  ```
  @age now => ('I am', 26, 'years old')



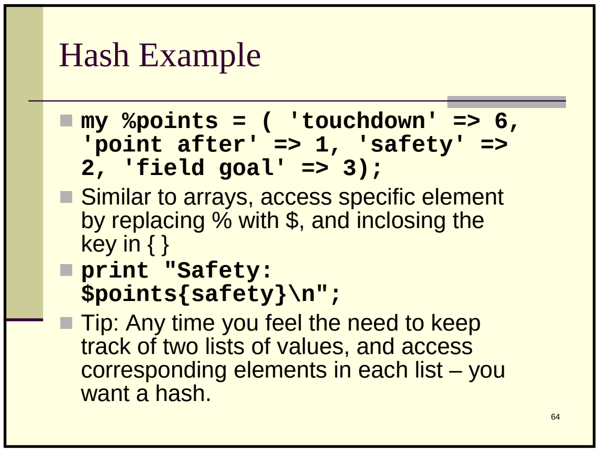
Arrays and Scalars

- `my $foo = 3;`
- `my @foo = (43.3, 10, 5.12, 'a');`
- $foo and @foo are *completely unrelated*
- In fact, $foo has nothing to do with $foo[2];
- "This may seem a bit weird, but that's okay, because it *is* weird."
    - Programming Perl, pg. 54



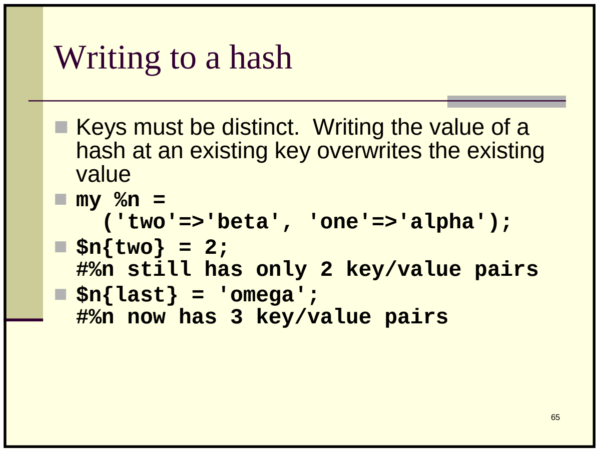
Hashes

- Also known as associative arrays
- a list of key/value pairs.
- Similar to arrays, but 'indices' can be any scalar value, not just integers.
    - both the keys and values can be any scalar value, including multiple types in the same hash.
- Used to maintain a list of corresponding values.
- Hash names start with a %
    - remainder follows same rules as array & scalar



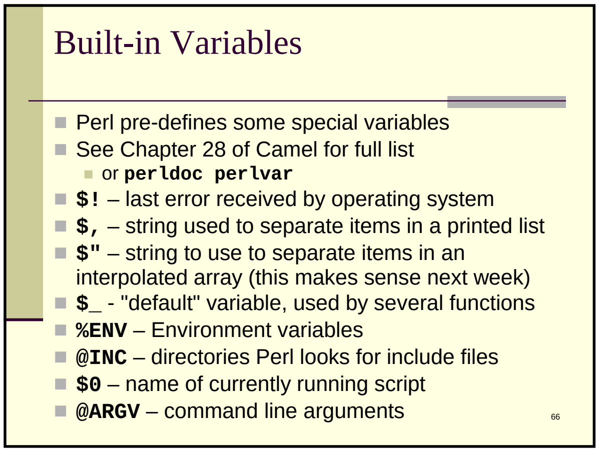
Hash Example

- `my %points = ( 'touchdown' => 6, 'point after' => 1, 'safety' => 2, 'field goal' => 3);`
- Similar to arrays, access specific element by replacing % with $, and inclosing the key in { }
- `print "Safety: $points{safety}\n";`
- Tip: Any time you feel the need to keep track of two lists of values, and access corresponding elements in each list – you want a hash.



Writing to a hash

- Keys must be distinct. Writing the value of a hash at an existing key overwrites the existing value
-
  ```
  my %n =
    ('two'=>'beta', 'one'=>'alpha');
  ```
-
  ```
  $n{two} = 2;
  #%n still has only 2 key/value pairs
  ```
-
  ```
  $n{last} = 'omega';
  #%n now has 3 key/value pairs
  ```



Built-in Variables

- Perl pre-defines some special variables
- See Chapter 28 of Camel for full list
  - or **`perldoc perlvar`**
- **`$!`** – last error received by operating system
- **`$,`** – string used to separate items in a printed list
- **`$"`** – string to use to separate items in an interpolated array (this makes sense next week)
- **`$_`** - "default" variable, used by several functions
- **`%ENV`** – Environment variables
- **`@INC`** – directories Perl looks for include files
- **`$0`** – name of currently running script
- **`@ARGV`** – command line arguments

Reading from the keyboard

- The "diamond" operator: **`<>`**
- Encloses the filehandle you want to read from. For now, the only filehandle is STDIN:
- **`my $line = <STDIN>;`**
  - Reads next line from standard input (ie, the keyboard), and stores it in $line.

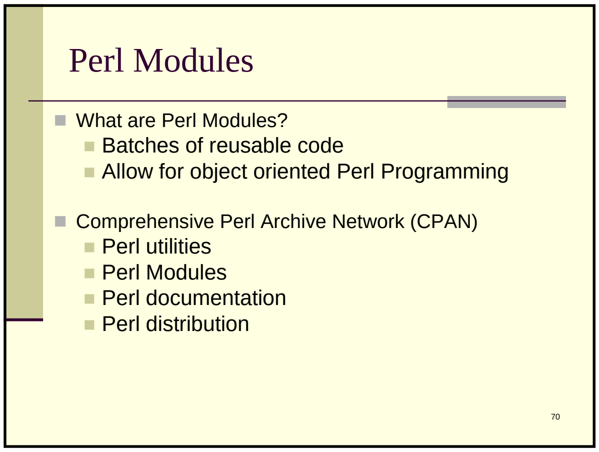
chomp LIST

- When you read a line, the *entire* line is read – including the trailing "\n".
- If you don't want the "\n", make sure you **`chomp`** your string.
- **`chomp`** takes a list of strings. If any string passed in contains a trailing newline, that newline is removed.
  - Operates directly on argument
  - Does not return the "chomp"ed string
    - Returns the number of newlines removed
  - Does not remove multiple newlines from a single string.

chomp examples

-
  ```
  my $input = <STDIN>;
  chomp $input;
  ```
- This is so common that there's a shorthand idiom:
  - **`chomp (my $input = <STDIN>);`**

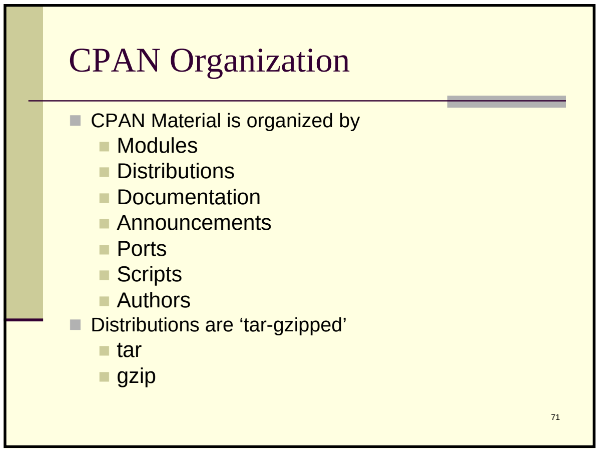
-
  ```
  my @strings =
    ("foo\n", "bar", "baz\n\n");
  chomp @strings;
  ```
  - @strings ➔("foo", "bar", "baz\n");
- There exists a similar function: **`chop`**
  - removes last character of a string, regardless of what it is.



Perl Modules

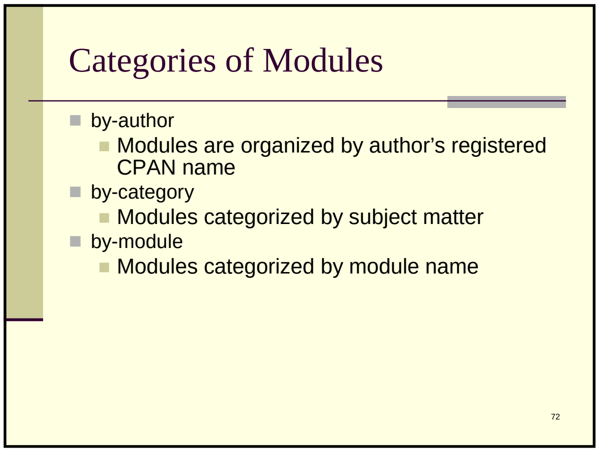
- What are Perl Modules?
  - Batches of reusable code
  - Allow for object oriented Perl Programming

- Comprehensive Perl Archive Network (CPAN)
  - Perl utilities
  - Perl Modules
  - Perl documentation
  - Perl distribution



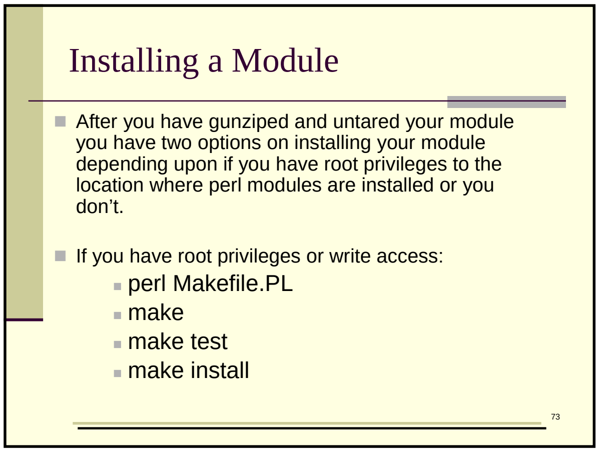
CPAN Organization

- CPAN Material is organized by
  - Modules
  - Distributions
  - Documentation
  - Announcements
  - Ports
  - Scripts
  - Authors
- Distributions are ‘tar-gzipped’
  - tar
  - gzip



Categories of Modules

- by-author
  - Modules are organized by author’s registered CPAN name
- by-category
  - Modules categorized by subject matter
- by-module
  - Modules categorized by module name



Installing a Module

- After you have gunziped and untared your module you have two options on installing your module depending upon if you have root privileges to the location where perl modules are installed or you don't.
- If you have root privileges or write access:
  - perl Makefile.PL
  - make
  - make test
  - make install