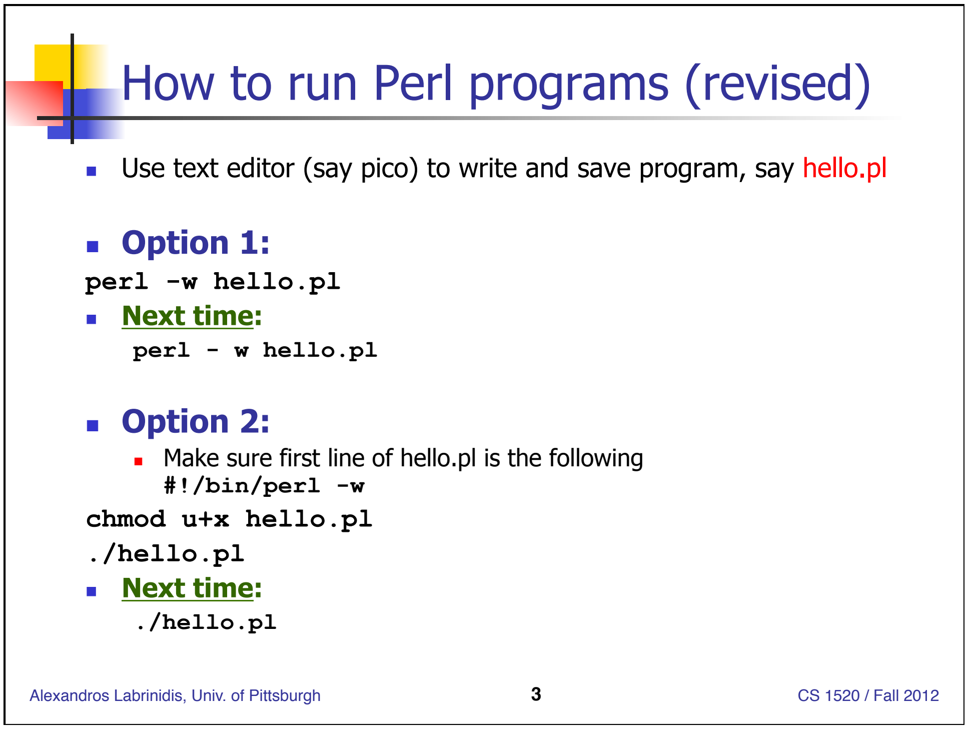
CS 1520 / Fall 2012
Programming Languages for Web Applications

01 – Introduction to Perl

Alexandros Labrinidis
University of Pittsburgh

What is Perl

- **P**ractical **E**xtraction and **R**eporting **L**anguage
  - 1987
  - Larry Wall
  - roots in unix applications (awk/sed)
  - http://history.perl.org
- Uses of Perl:
  - EVERYWHERE :-)
  - Scripts for repeated tasks
  - Powerful regular expression matching
    - Text processing
  - Web programming



How to run Perl programs (revised)

- Use text editor (say pico) to write and save program, say hello.pl

- **Option 1:**

```
perl -w hello.pl
```

- **Next time:**

```
 perl - w hello.pl
```

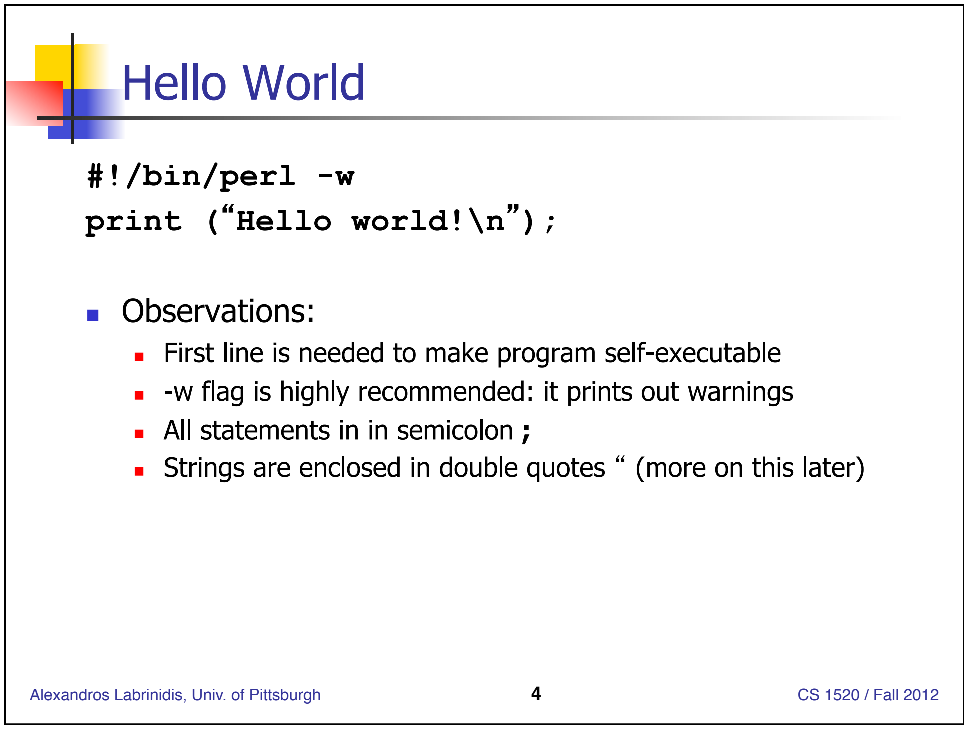
- **Option 2:**
  - Make sure first line of hello.pl is the following

```
#!/bin/perl -w
```

```
chmod u+x hello.pl
./hello.pl
```

- **Next time:**

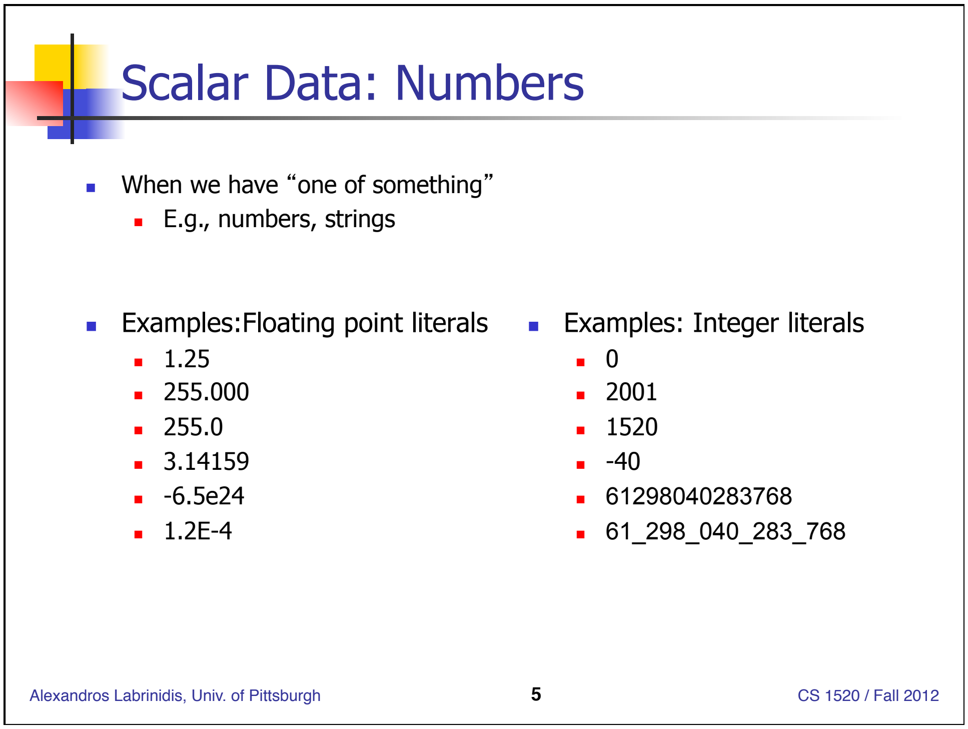
```
 ./hello.pl
```



Hello World

```
#!/bin/perl -w
print (“Hello world!\n”);
```

- Observations:
  - First line is needed to make program self-executable
  - -w flag is highly recommended: it prints out warnings
  - All statements in in semicolon **;**
  - Strings are enclosed in double quotes “ (more on this later)

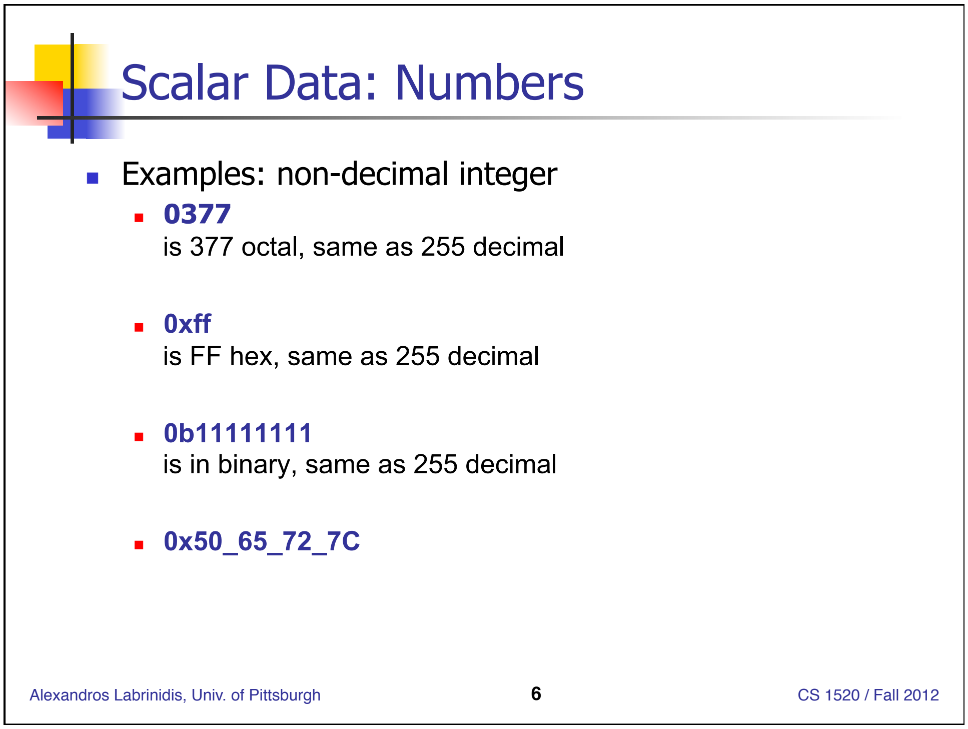


Scalar Data: Numbers

- When we have “one of something”
  - E.g., numbers, strings

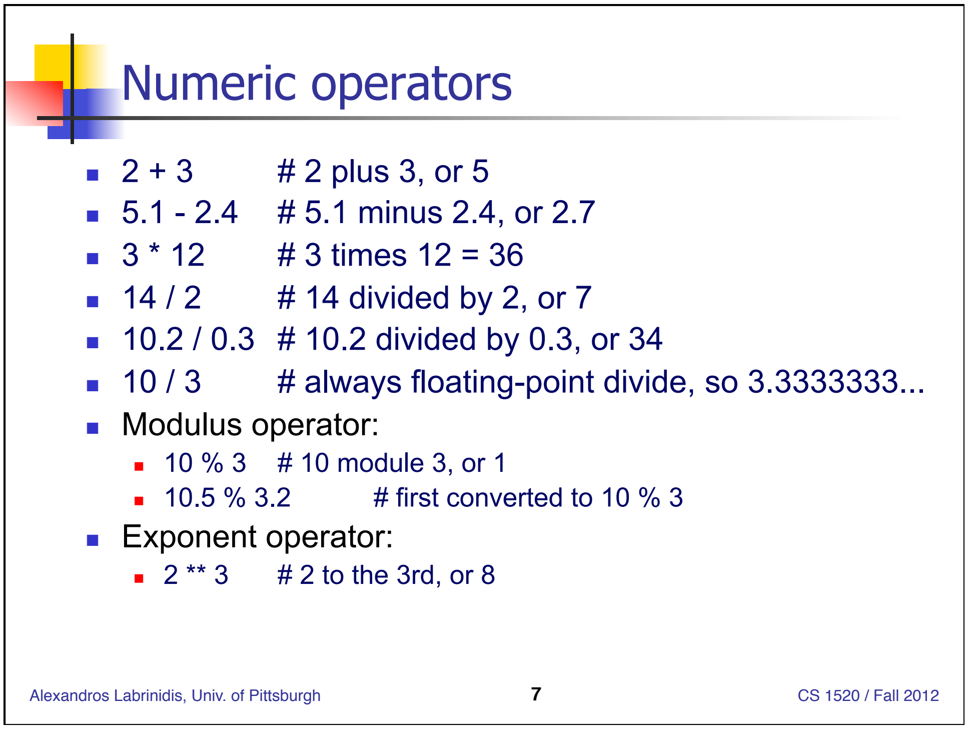
- Examples:Floating point literals
  - 1.25
  - 255.000
  - 255.0
  - 3.14159
  - -6.5e24
  - 1.2E-4

- Examples: Integer literals
  - 0
  - 2001
  - 1520
  - -40
  - 61298040283768
  - 61_298_040_283_768



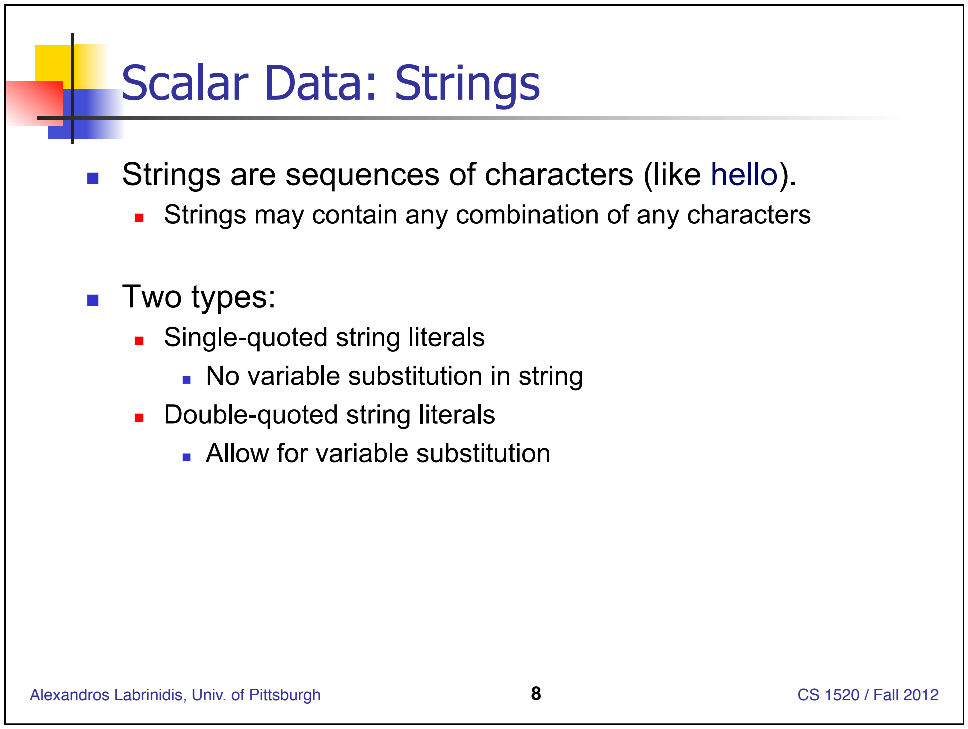
Scalar Data: Numbers

- Examples: non-decimal integer
  - **0377**
    is 377 octal, same as 255 decimal
  - **0xff**
    is FF hex, same as 255 decimal
  - **0b11111111**
    is in binary, same as 255 decimal
  - **0x50_65_72_7C**



Numeric operators

- 2 + 3 # 2 plus 3, or 5
- 5.1 - 2.4 # 5.1 minus 2.4, or 2.7
- 3 * 12 # 3 times 12 = 36
- 14 / 2 # 14 divided by 2, or 7
- 10.2 / 0.3 # 10.2 divided by 0.3, or 34
- 10 / 3 # always floating-point divide, so 3.3333333...
- Modulus operator:
  - 10 % 3 # 10 module 3, or 1
  - 10.5 % 3.2 # first converted to 10 % 3
- Exponent operator:
  - 2 ** 3 # 2 to the 3rd, or 8



Scalar Data: Strings

- Strings are sequences of characters (like hello).
  - Strings may contain any combination of any characters
- Two types:
  - Single-quoted string literals
    - No variable substitution in string
  - Double-quoted string literals
    - Allow for variable substitution

Single-quoted Strings

- 'fred'   # those four characters: f, r, e, and d
- 'barney'   # those six characters
- ''   # the null string (no characters)
- 'Don\'t let an apostrophe end this string prematurely!’
- 'the last character of this string is a backslash: \\’
- 'hello\n'   # hello followed by backslash followed by n
- 'hello
  there'   # hello, newline, there (11 characters total)
- ’\’ \\'   # single quote followed by backslash
- Note that the \n within a single-quoted string is not interpreted as a newline, but as the two characters backslash and n. Only when the backslash is followed by another backslash or a single quote does it have special meaning.

Double-quoted strings

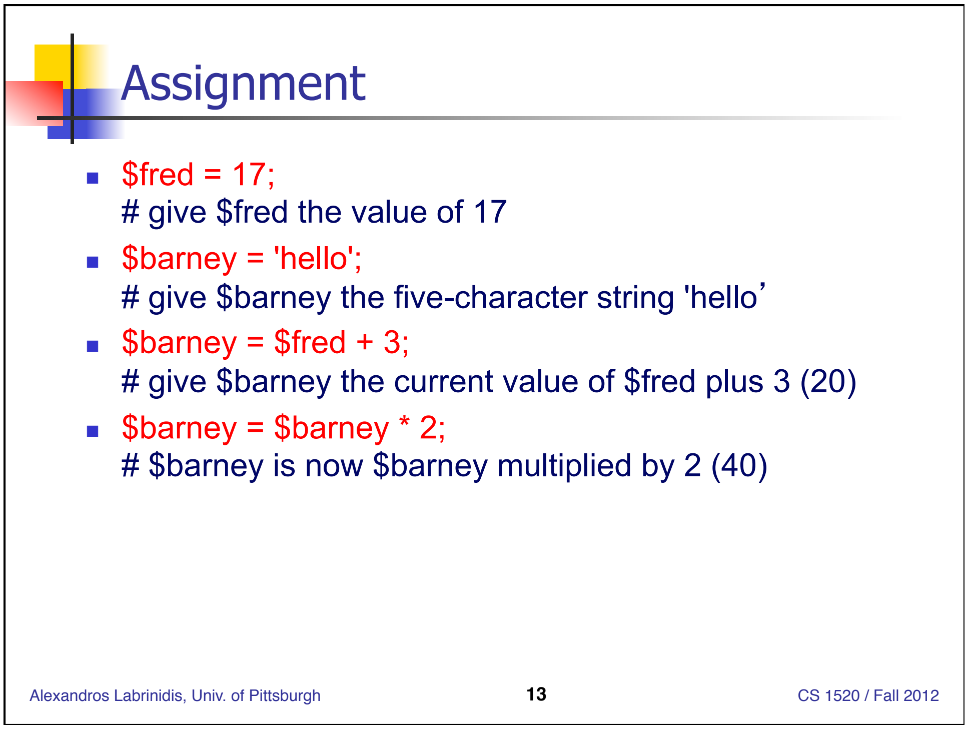
- "barney"   # just the same as 'barney’
- "hello world\n"   # hello world, and a newline
- "The last character of this string is a quote mark: \"”
- "coke\tsprite"   # coke, a tab, and sprite

String operators

- Concatenation
  - "hello" . "world"   # same as "helloworld”
  - "hello" . ' ' . "world" # same as 'hello world’
  - 'hello world' . ”\n"   # same as "hello world\n”
- String repetition
  - "fred" x 3   # is "fredfredfred”
  - "barney" x (4+1) # is "barney" x 5, or "barneybarneybarneybarneybarney”
  - 5 x 4   # is really "5" x 4, which is "5555"

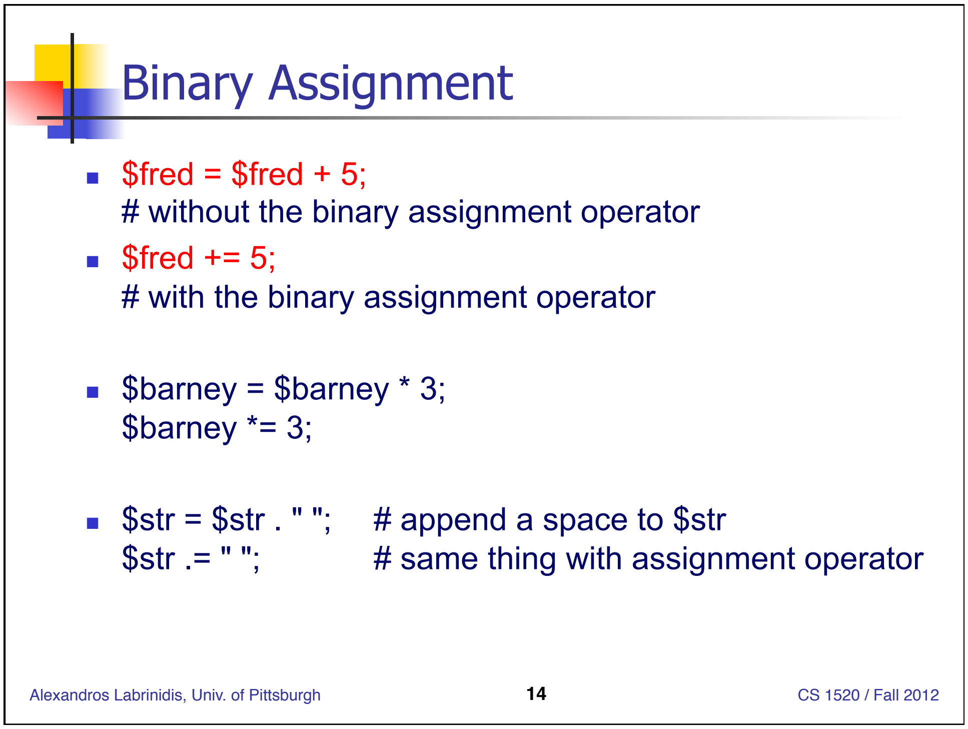
Scalar Variables

- A *variable* is a name for a container that holds one or more values
  - Unlike other languages, there is no variable definition/declaration in Perl (unless you invoke “strict”)
- Example variable names:
  - $a_very_long_variable_that_ends_in_1
  - $a_very_long_variable_that_ends_in_2
  - $Fred is different from $fred
  - $is_it_better_with_underscores
  - $orIsItBetterWithCaps



Assignment

- $fred = 17;
  # give $fred the value of 17
- $barney = 'hello';
  # give $barney the five-character string 'hello’
- $barney = $fred + 3;
  # give $barney the current value of $fred plus 3 (20)
- $barney = $barney * 2;
  # $barney is now $barney multiplied by 2 (40)

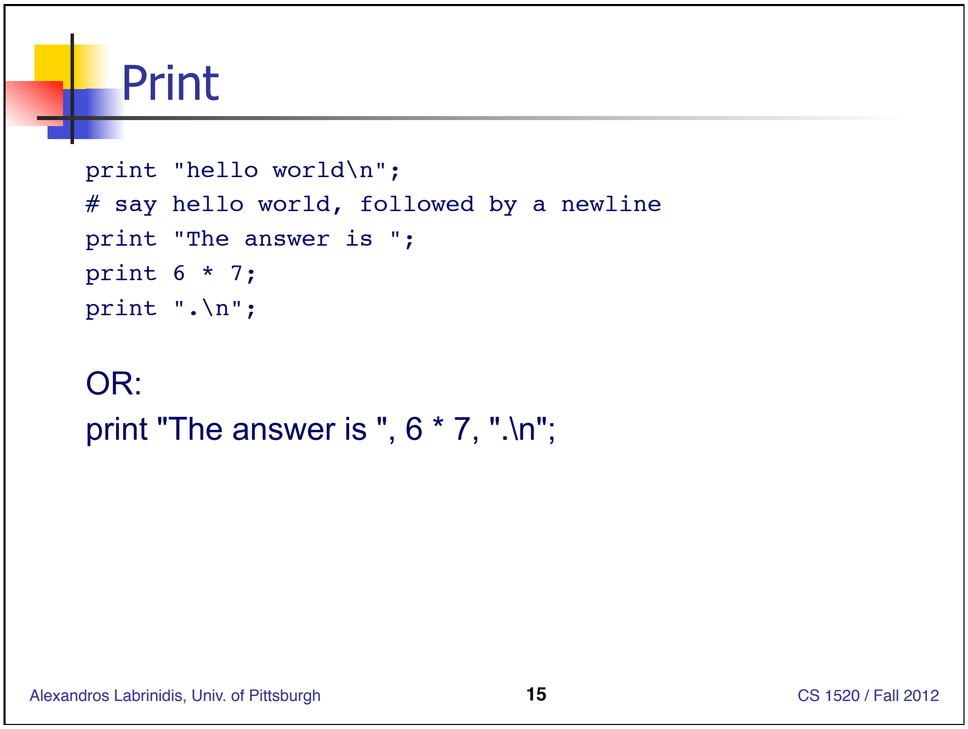


Binary Assignment

- $fred = $fred + 5;
  # without the binary assignment operator
- $fred += 5;
  # with the binary assignment operator

- $barney = $barney * 3;
  $barney *= 3;

-
  ```
  $str = $str . " ";    # append a space to $str
  $str .= " ";          # same thing with assignment operator
  ```

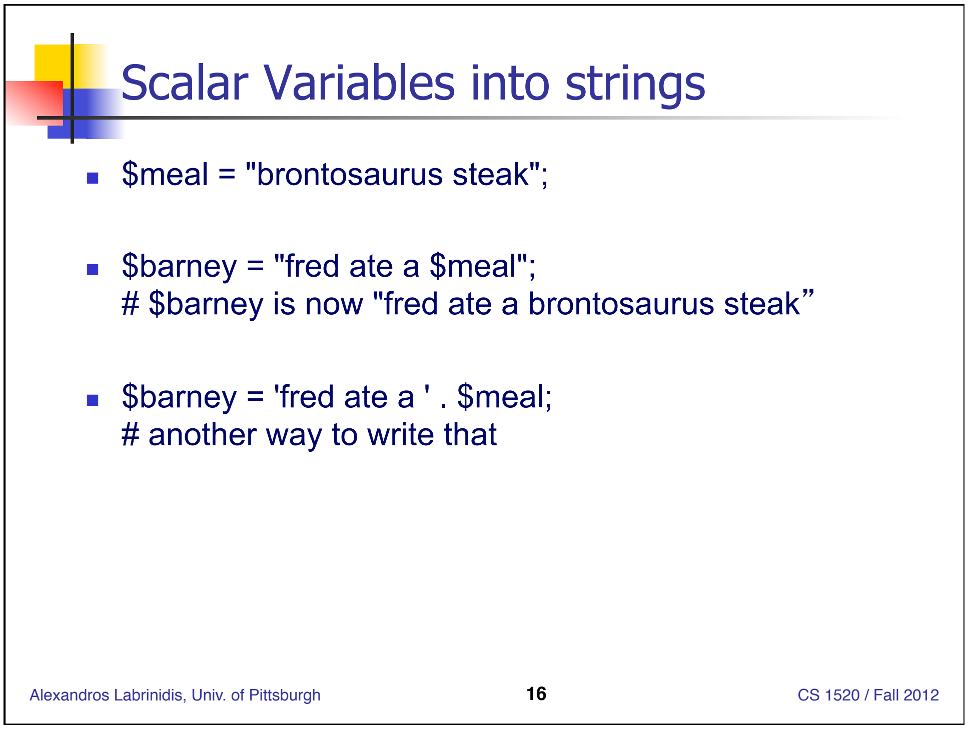


Print

```
print "hello world\n";
# say hello world, followed by a newline
print "The answer is ";
print 6 * 7;
print ".\n";
```

OR:

```
print "The answer is ", 6 * 7, ".\n";
```



Scalar Variables into strings

- $meal = "brontosaurus steak";
- $barney = "fred ate a $meal";
  # $barney is now "fred ate a brontosaurus steak”
- $barney = 'fred ate a ' . $meal;
  # another way to write that

Lists and Arrays in Perl

- **Q:** What is a List or Array?
- **A:** a **list** is ordered scalar data
- **A:** an **array** is a variable that holds a list

- Example - list literals:
  - (1, 2, 3) #array of three values: 1, 2, and 3
  - (“fred”, 4, 5) #array of three values: “fred”, 4, and 5
  - ($a, 42) #two values: current value of $a and 42
  - ($a+$b, $c+$d) #also two values
  - () #the empty list (no elements)
  - (1 .. 5) #same as (1, 2, 3, 4, 5)

More examples - arrays

- @a = (“rachel”, “phoebe”, “chandler”, “ross”, “monica”, “joey”);

- Shortcut:
- @a = **qw**(rachel phoebe chandler ross monica joey);
  - **qw** stands for **quote word**: it creates a list from the nonwhitespace parts within the parentheses

- @b = qw(mary 2 5 had a little 4 lamb 6);

- **Q:** what will the following do?

```
print (“the answer is “,@a,”\n”);
```

Arrays - Assignment

- @rachel = (1, 2, 3);
  #the *rachel* array is a 3-element list literal

- @monica = @rachel;
  #the *rachel* array is copied to the *monica* array

- @joey = 1;
  # 1 is automatically converted to (1)

- @ross = qw(one two three);
  # equivalent to @ross = ("one", "two", "three");

Arrays - Assignment (RHS)

- @phoebe = qw(smelly cat);
  #equivalent to @phoebe = ("smelly", "cat");

- @chandler = ("here", "is", @phoebe, 5, 6);
  #chandler becomes ("here", "is", "smelly", "cat", 5, 6);

- @chandler = (42, @chandler);
  #puts 42 in front of chandler

- @chandler = (@chandler, "last");
  #puts "last" at the end of chandler

Arrays - Assignment (LHS)

- ($a, $b, $c) = (1, 2, 3)
  #give value of 1 to $a, 2 to $b, 3 to $c

- ($a, $b) = ($b, $a)
  #swap $a and $b

- ($d, @ross) = ($a, $b, $c)
  #give $a to $d, and ($b, $c) to @ross

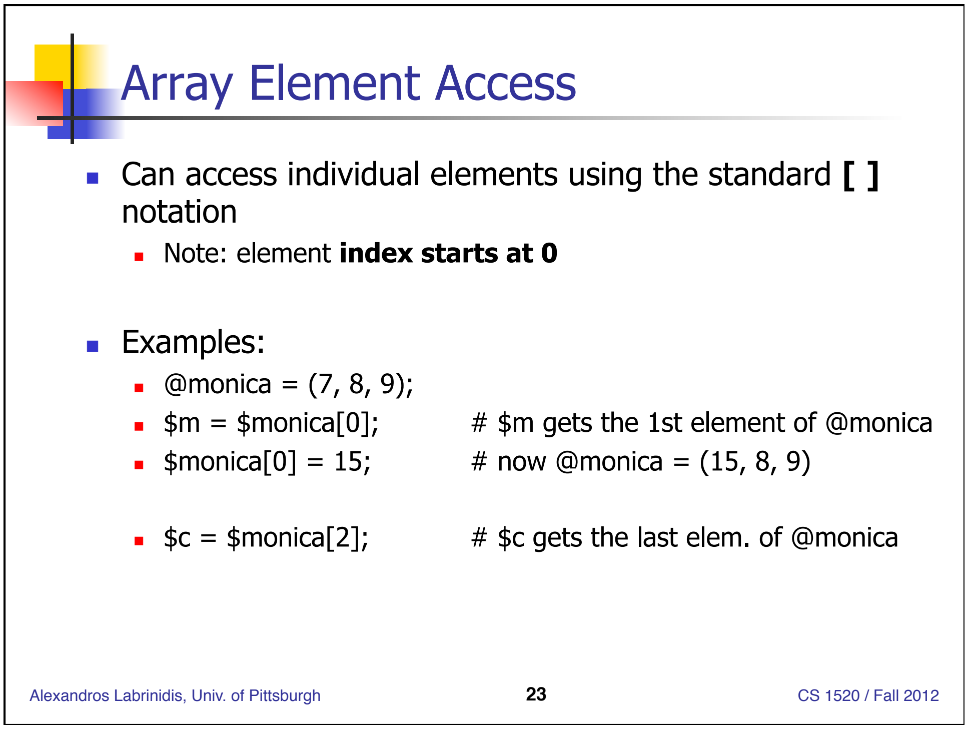
- ($f, @ross) = @ross;
  #remove first element of @ross and give it to $f
  #this makes $f = $b, and @ross = ($c)

Scalar and List context

- Perl will do automatic "conversion" of an array, depending on the context.

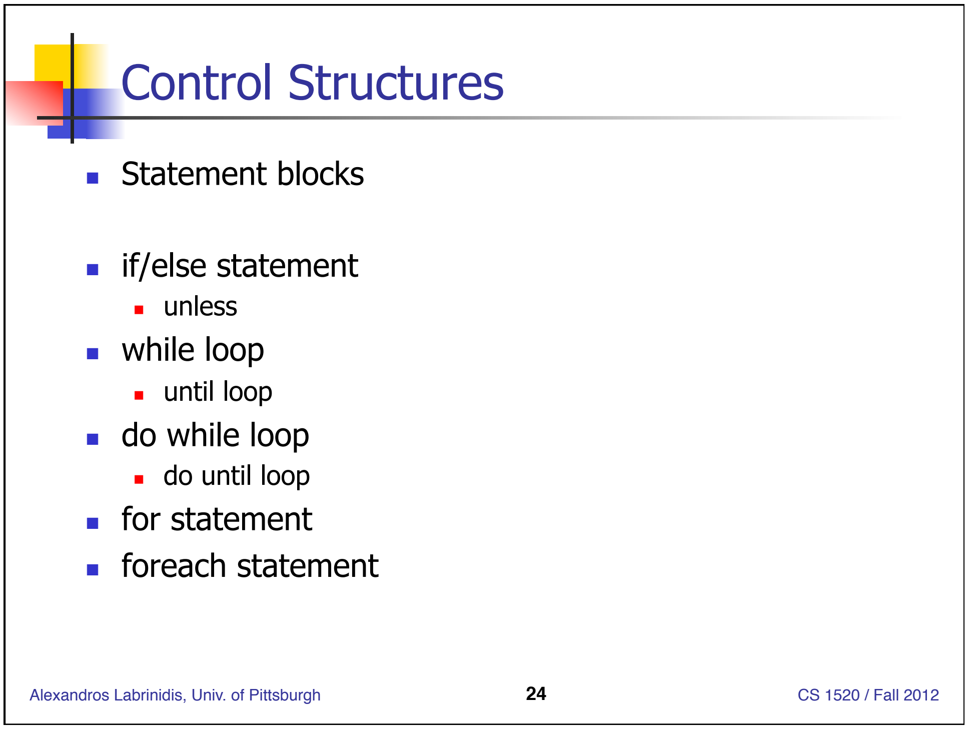
- **List context**: normal
- **Scalar context**: when the array must be "squeezed" into a scalar variable.
  - In this case, the length of the array is returned!

- Examples:
  - @barney = @fred; # all of @fred is copied to @barney
  - ($a) = @fred; # $a gets the first element of @fred
  - $a = @fred; # $a gets the **length of @fred**



Array Element Access

- Can access individual elements using the standard **[ ]** notation
  - Note: element **index starts at 0**
- Examples:
  - @monica = (7, 8, 9);
  - $m = $monica[0];   # $m gets the 1st element of @monica
  - $monica[0] = 15;   # now @monica = (15, 8, 9)
  - $c = $monica[2];   # $c gets the last elem. of @monica



Control Structures

- Statement blocks
- if/else statement
  - unless
- while loop
  - until loop
- do while loop
  - do until loop
- for statement
- foreach statement

Statement blocks

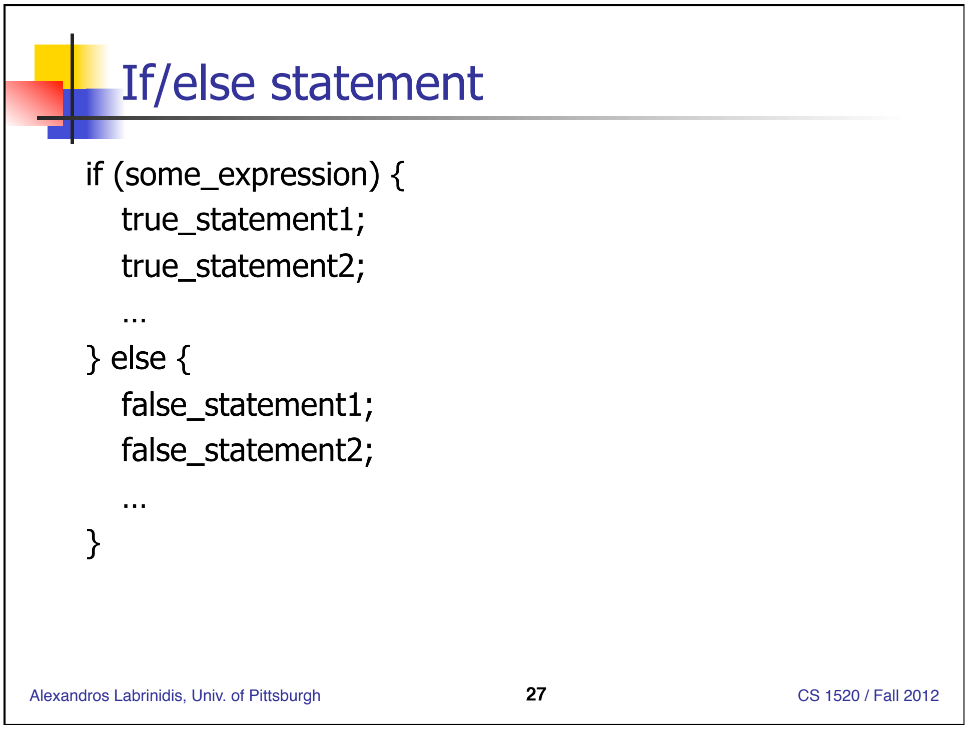
```
{
   first_statement;
   second_statement;
   third_statement;
   …
   last_statement;
}
```

Control Structures: if/then

- **if/else**

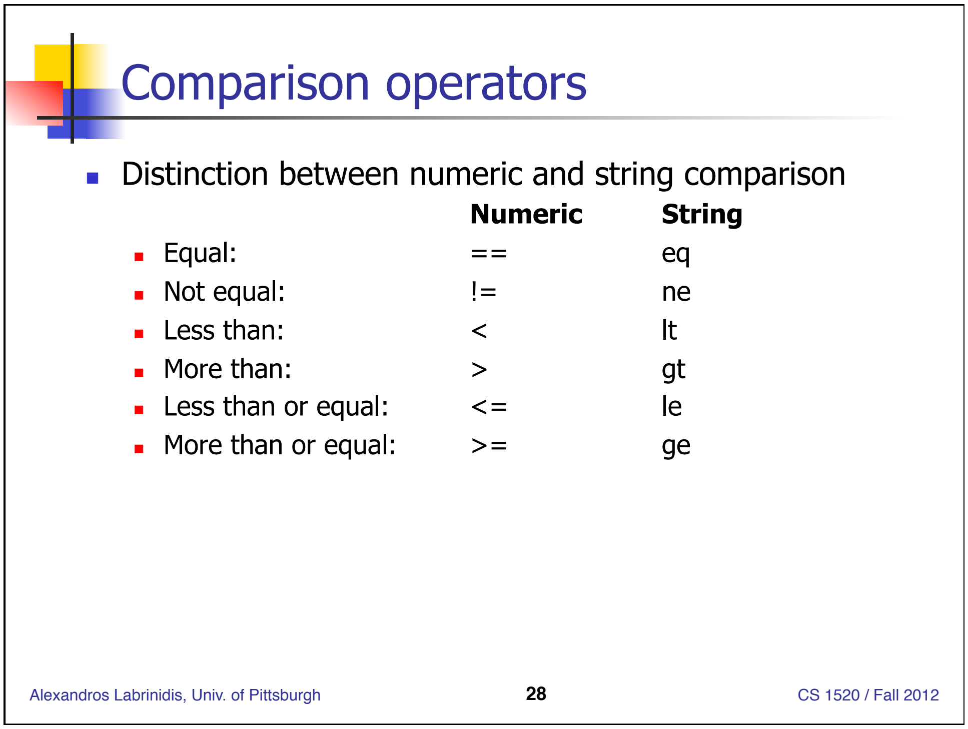
  -
    ```
    if ($a > 5) {
      #do nothing
    } else {
      $a++;
    }
    ```
- **unless**

  -
    ```
    Unless ($a>5) {
      $a++;
    }
    ```



If/else statement

```
if (some_expression) {
   true_statement1;
   true_statement2;
   …
} else {
   false_statement1;
   false_statement2;
   …
}
```



Comparison operators

- Distinction between numeric and string comparison

	Numeric	String
Equal:	==	eq
Not equal:	!=	ne
Less than:	<	lt
More than:	>	gt
Less than or equal:	<=	le
More than or equal:	>=	ge

If/elsif/else statement

```
if (first_expression) {
   true_first_statement1;
   true_first_statement2;
   …
} elsif (second_expression) {
   true_second_statement1;
   true_second_statement2;
} else {
   all_false_statement1;
   all_false_statement2;
   …
}
```

Control Structures: while

- **while loop**

  -
    ```
    while ($a > 5) {
      #do something
    }
    ```
- **until**

  -
    ```
    until ($a <= 5) {
      $a++;
    }
    ```

Control Structures: do/while

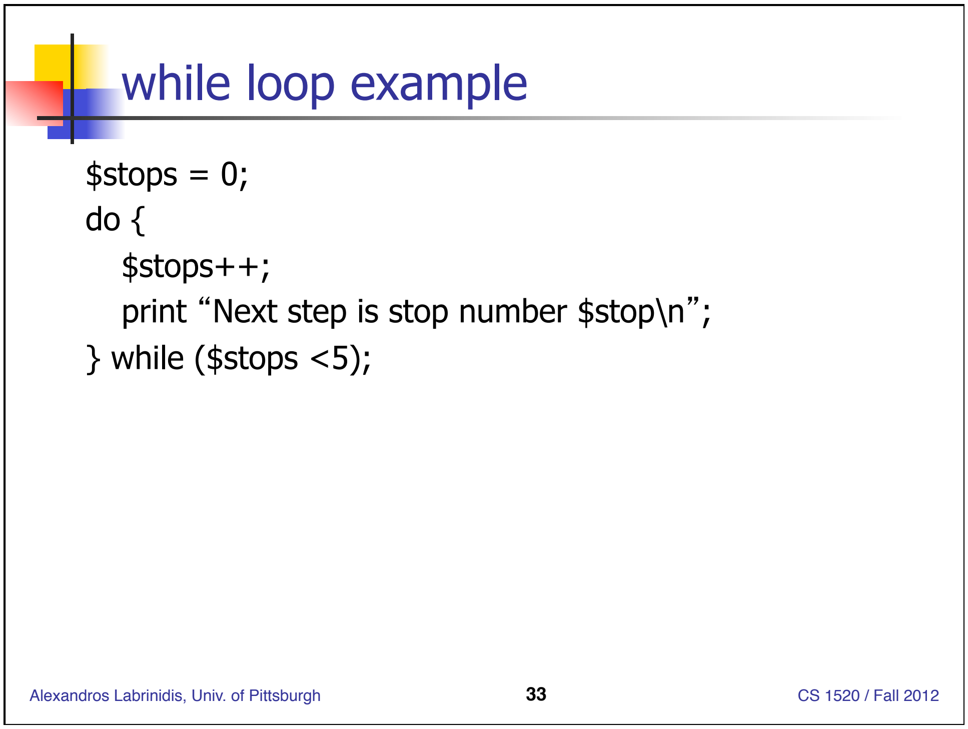
- **do/while loop**

  -
    ```
    do {
      #do something
    } while ($a > 5);
    ```
- **do/until loop**

  -
    ```
    do {
      #do something
    } until ($a <= 5);
    ```

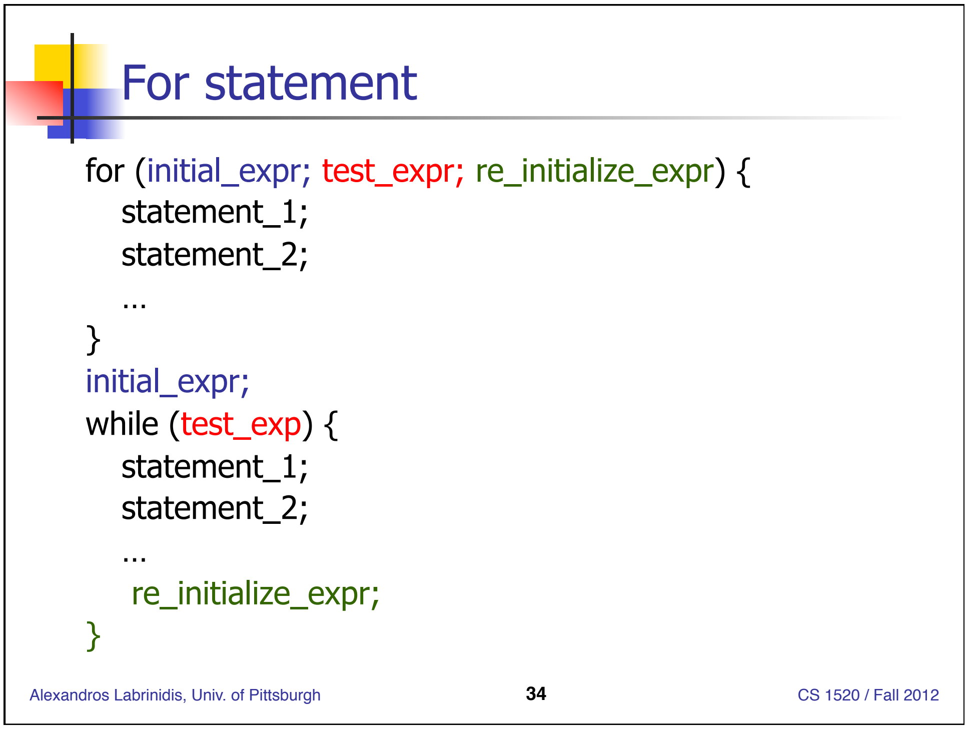
while and do/while loops

```
while (some_expression) {
   statement_1;
   statement_2;
   …
}

do {
   statement_1;
   statement_2;
   …
} while (some_expression);
```



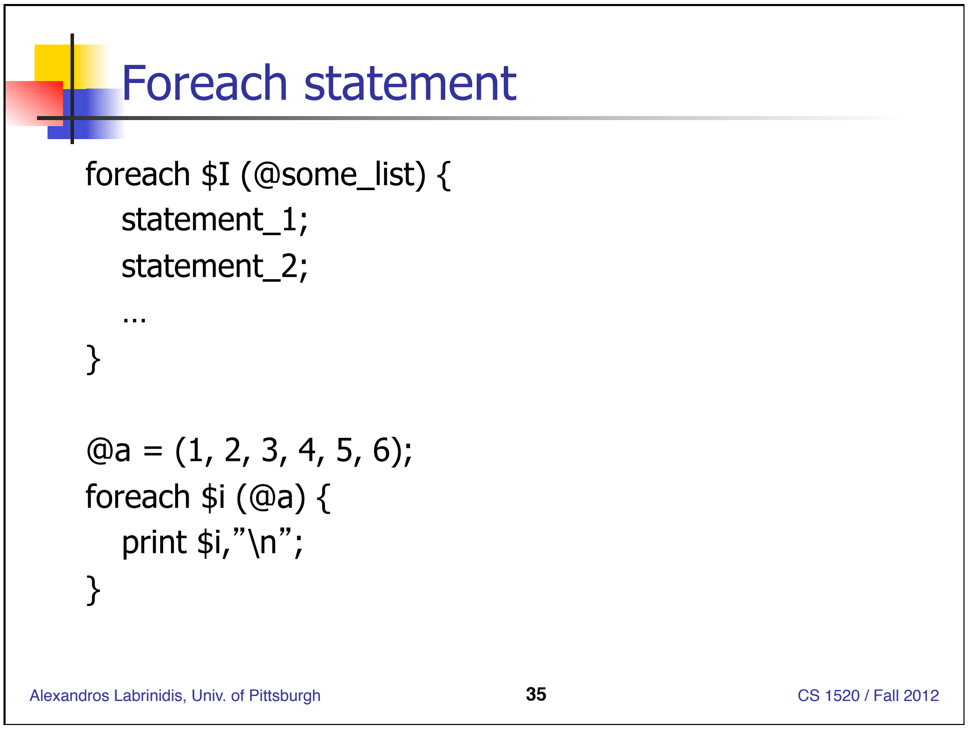
while loop example

```
$stops = 0;
do {
   $stops++;
   print “Next step is stop number $stop\n”;
} while ($stops <5);
```



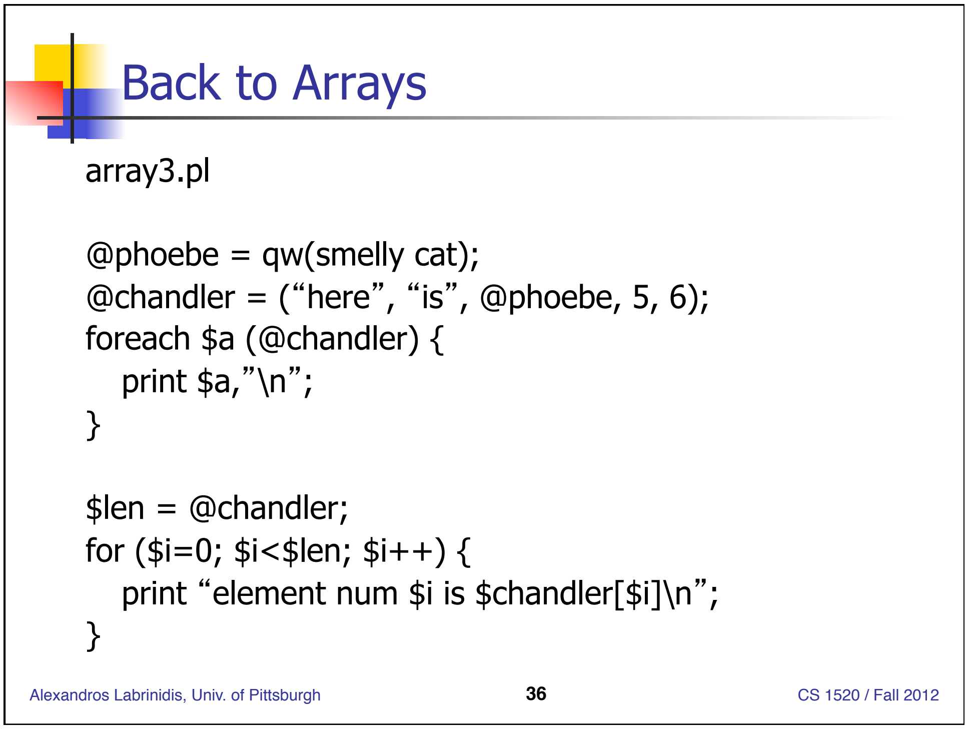
For statement

```
for (initial_expr; test_expr; re_initialize_expr) {
   statement_1;
   statement_2;
   …
}
initial_expr;
while (test_exp) {
   statement_1;
   statement_2;
   …
    re_initialize_expr;
}
```



Foreach statement

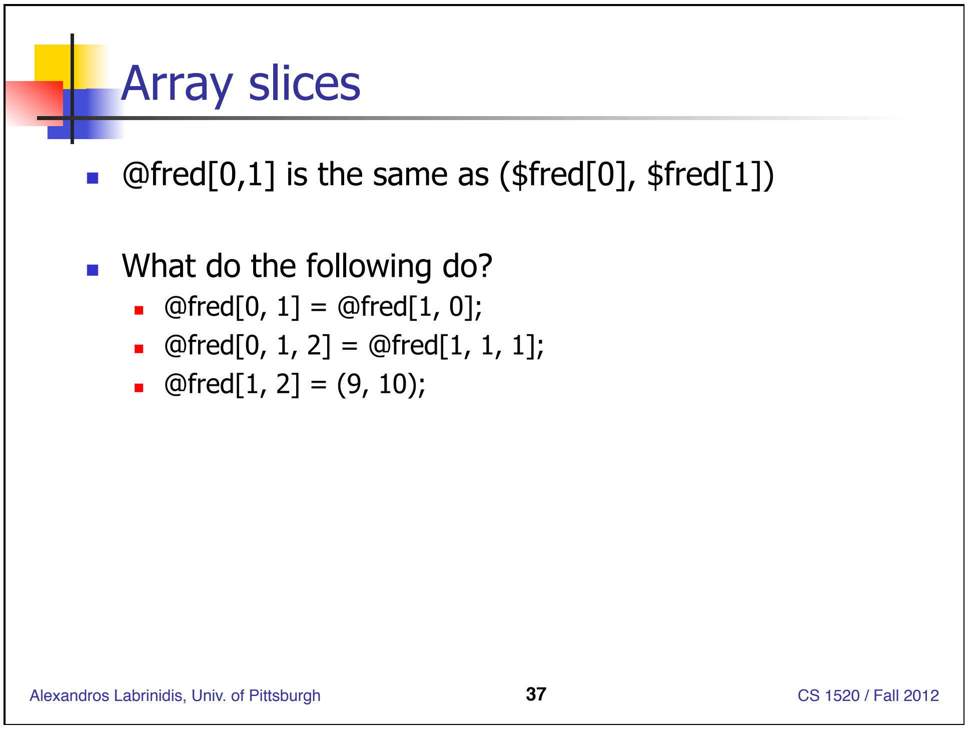
```
foreach $I (@some_list) {
   statement_1;
   statement_2;
   …
}

@a = (1, 2, 3, 4, 5, 6);
foreach $i (@a) {
   print $i,”\n”;
}
```



Back to Arrays

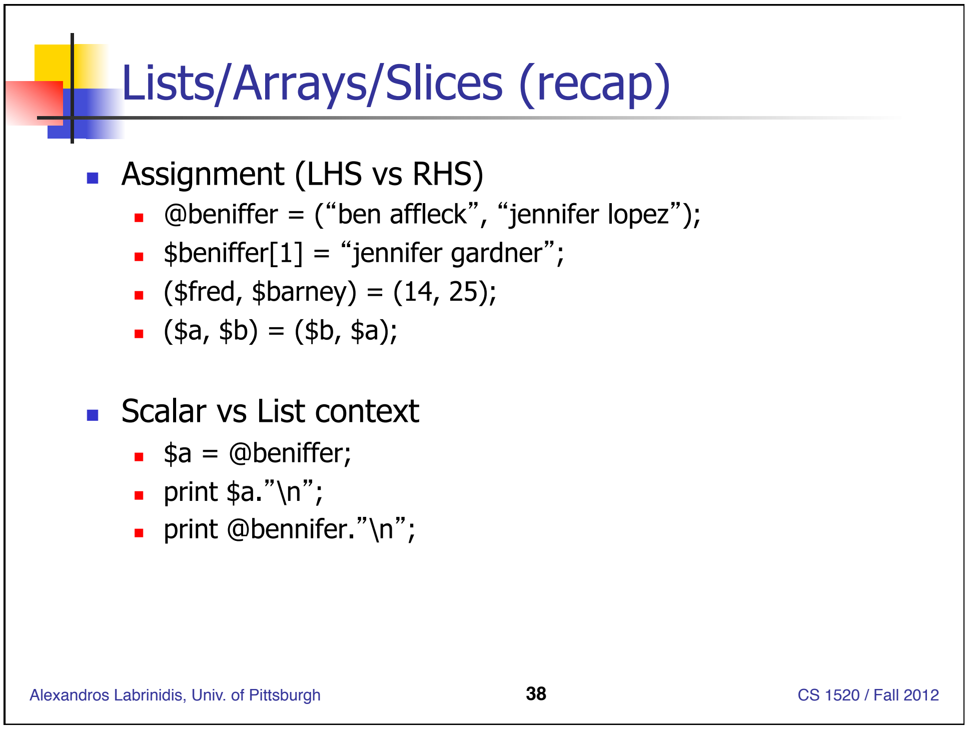
array3.pl

```
@phoebe = qw(smelly cat);
@chandler = (“here”, “is”, @phoebe, 5, 6);
foreach $a (@chandler) {
   print $a,”\n”;
}

$len = @chandler;
for ($i=0; $i<$len; $i++) {
   print “element num $i is $chandler[$i]\n”;
}
```



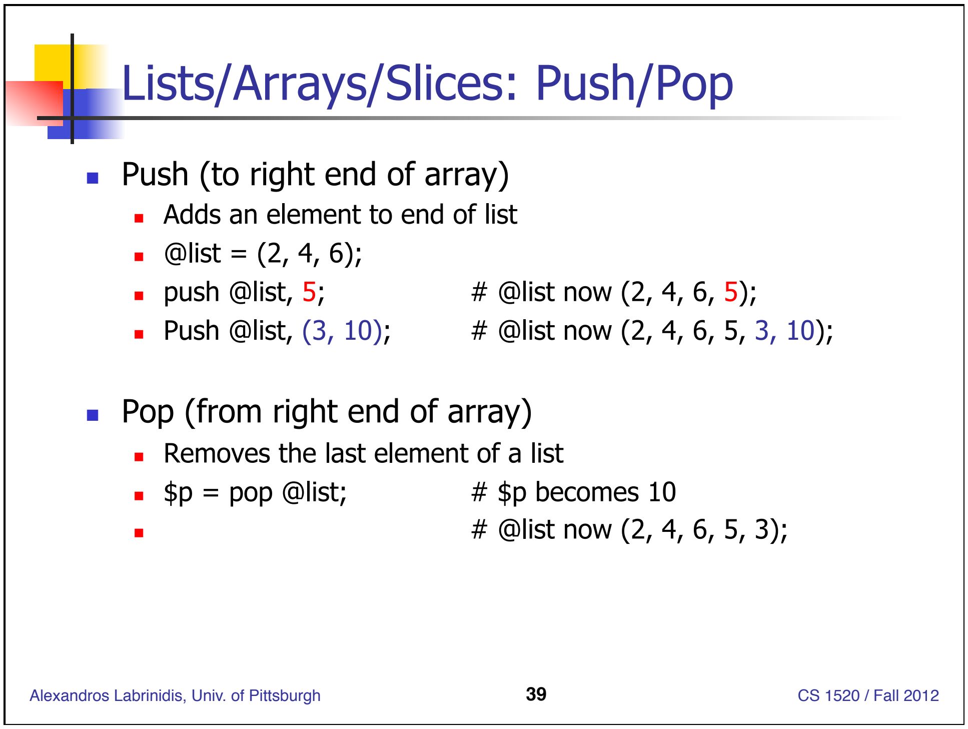
Array slices

- @fred[0,1] is the same as ($fred[0], $fred[1])
- What do the following do?
  - @fred[0, 1] = @fred[1, 0];
  - @fred[0, 1, 2] = @fred[1, 1, 1];
  - @fred[1, 2] = (9, 10);



Lists/Arrays/Slices (recap)

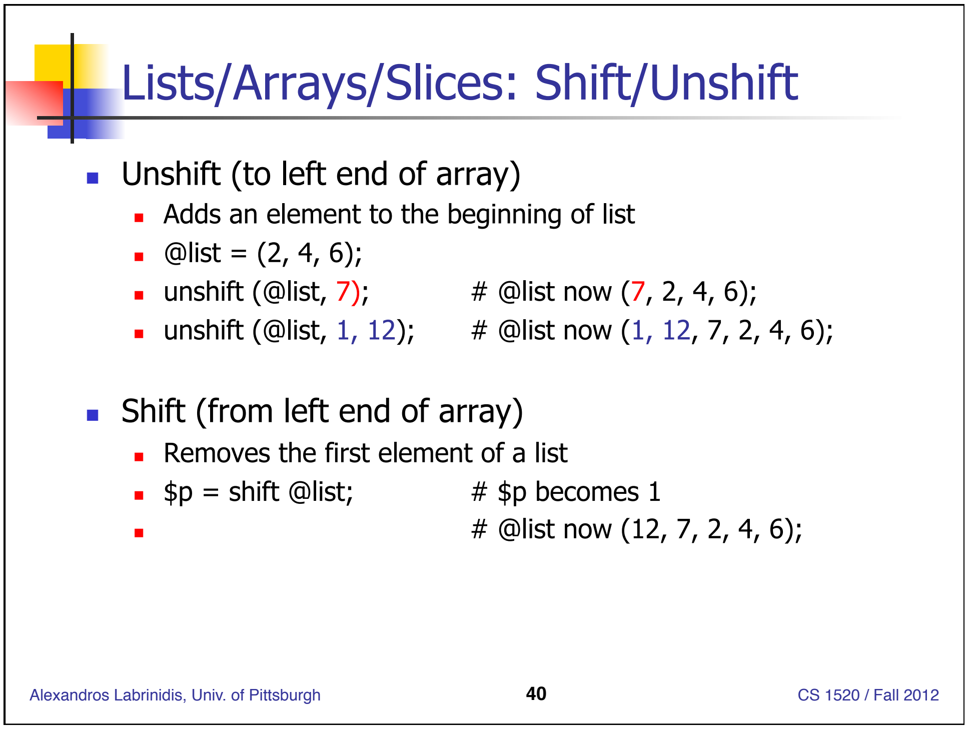
- Assignment (LHS vs RHS)
  - @beniffer = (“ben affleck”, “jennifer lopez”);
  - $beniffer[1] = “jennifer gardner”;
  - ($fred, $barney) = (14, 25);
  - ($a, $b) = ($b, $a);
- Scalar vs List context
  - $a = @beniffer;
  - print $a.”\n”;
  - print @bennifer.”\n”;



Lists/Arrays/Slices: Push/Pop

- Push (to right end of array)
  - Adds an element to end of list
  - @list = (2, 4, 6);
  - push @list, 5;   # @list now (2, 4, 6, 5);
  - Push @list, (3, 10);   # @list now (2, 4, 6, 5, 3, 10);

- Pop (from right end of array)
  - Removes the last element of a list
  - $p = pop @list;   # $p becomes 10
  -                         # @list now (2, 4, 6, 5, 3);



Lists/Arrays/Slices: Shift/Unshift

- Unshift (to left end of array)
  - Adds an element to the beginning of list
  - @list = (2, 4, 6);
  - unshift (@list, 7);   # @list now (7, 2, 4, 6);
  - unshift (@list, 1, 12); # @list now (1, 12, 7, 2, 4, 6);

- Shift (from left end of array)
  - Removes the first element of a list
  - $p = shift @list;   # $p becomes 1
  -                         # @list now (12, 7, 2, 4, 6);

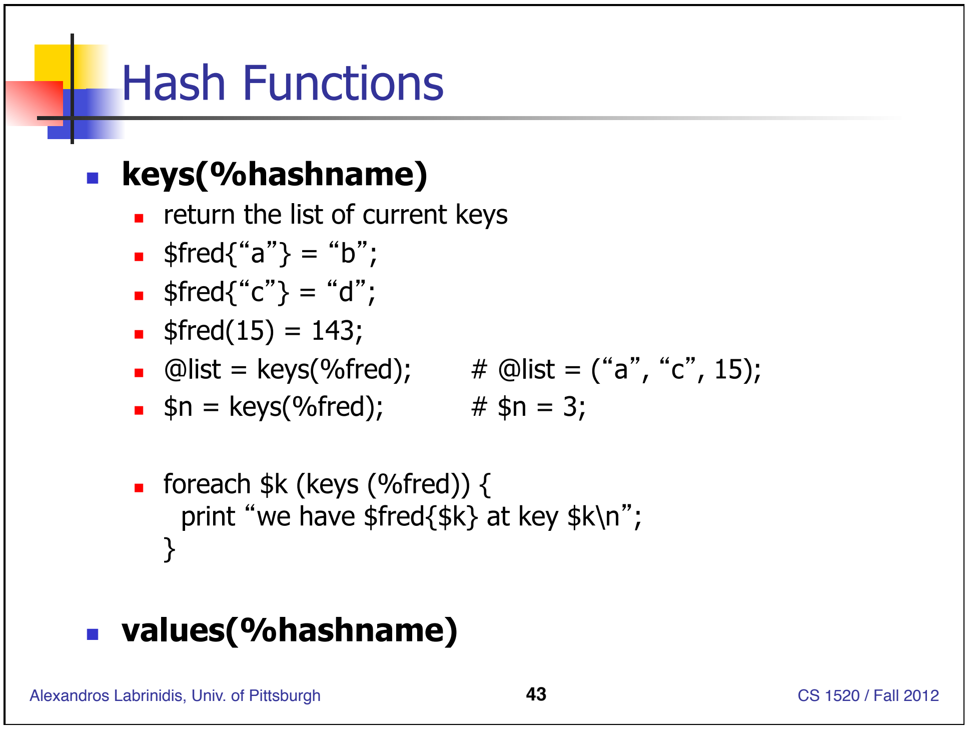
Lists/Arrays/Slices: Sort/Reverse

- Sort
  - sorts elements of an array
  - Example:
  - @x = qw(small medium large);
  - @y = sort (@x); # @x stays the same
  - # @y is (“large”, “medium”, “small”)
- Reverse
  - Reverses current order (i.e., NOT the reverse sort order)
  - @a = (10, 5, 12, 45);
  - @b = reverse @a; # @a stays the same
  - # @b is (45, 12, 5, 10)

Hashes

- Arrays are nice, index is always numeric
- **Q:** Would it not be nice to access arrays using strings?
- **A:** Yes.
- **A:** using Hashes!

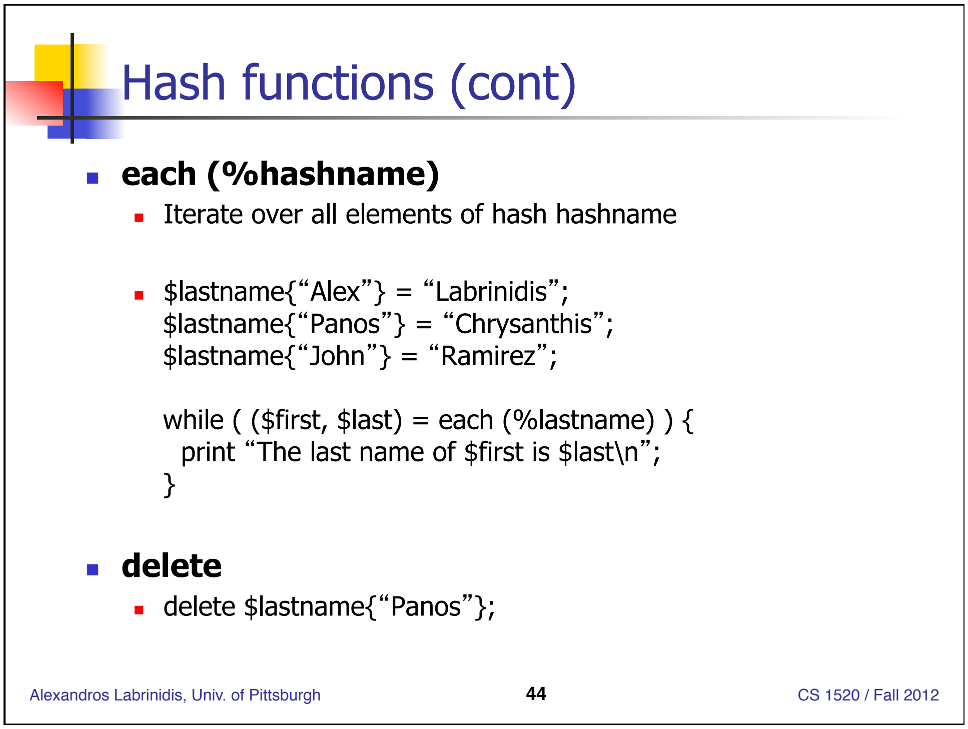
- **%**class is a hash
- Initialization:
  - $class{“CS1520”} = “SENSQ 5129”;
  - %class = (“CS1520” => “SENSQ 5129”,
“CS1555” => “SENSQ 5129”);
  - %class = (“CS1520”, “SENSQ 5129”, “CS1555”, “SENSQ 5129”);



Hash Functions

- **keys(%hashname)**
  - return the list of current keys
  - $fred{“a”} = “b”;
  - $fred{“c”} = “d”;
  - $fred(15) = 143;
  - @list = keys(%fred);   # @list = (“a”, “c”, 15);
  - $n = keys(%fred);   # $n = 3;

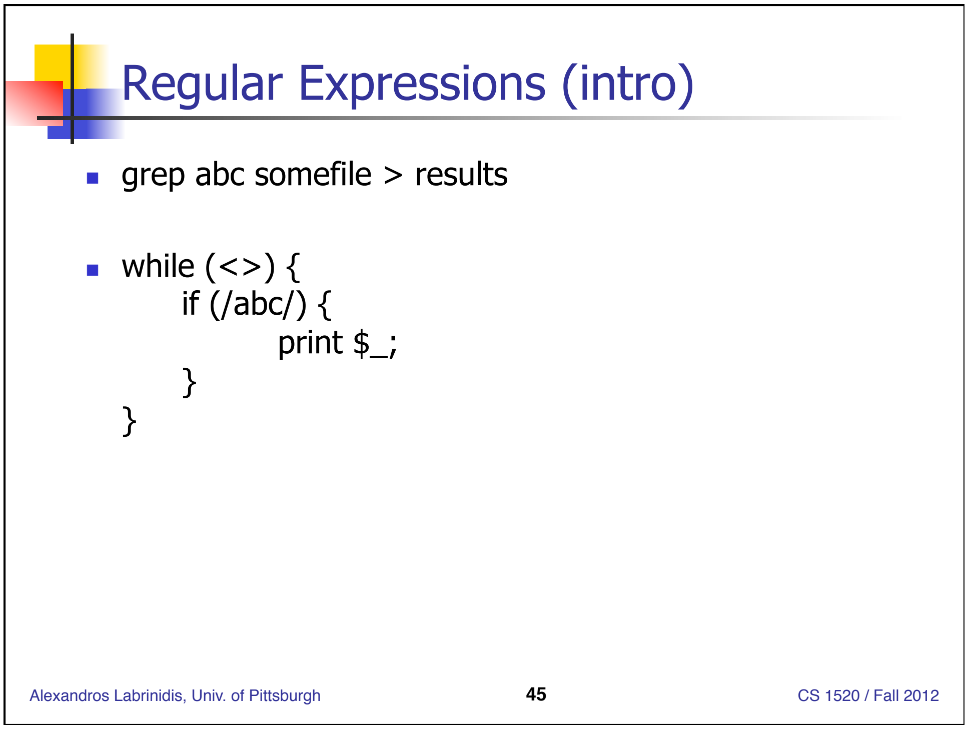
  -
    ```
    foreach $k (keys (%fred)) {
      print “we have $fred{$k} at key $k\n”;
    }
    ```
- **values(%hashname)**



Hash functions (cont)

- **each (%hashname)**
  - Iterate over all elements of hash hashname

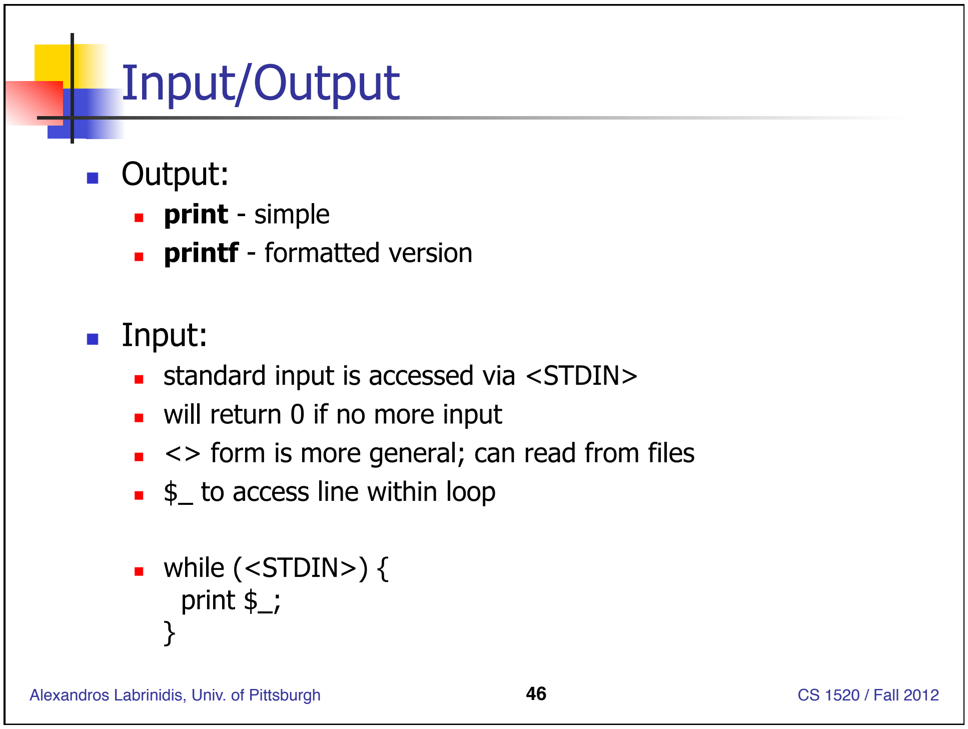
  -
    ```
    $lastname{“Alex”} = “Labrinidis”;
    $lastname{“Panos”} = “Chrysanthis”;
    $lastname{“John”} = “Ramirez”;

    while ( ($first, $last) = each (%lastname) ) {
      print “The last name of $first is $last\n”;
    }
    ```
- **delete**
  - delete $lastname{“Panos”};



Regular Expressions (intro)

- grep abc somefile > results

-
  ```
  while (<>) {
        if (/abc/) {
                  print $_;
        }
  }
  ```

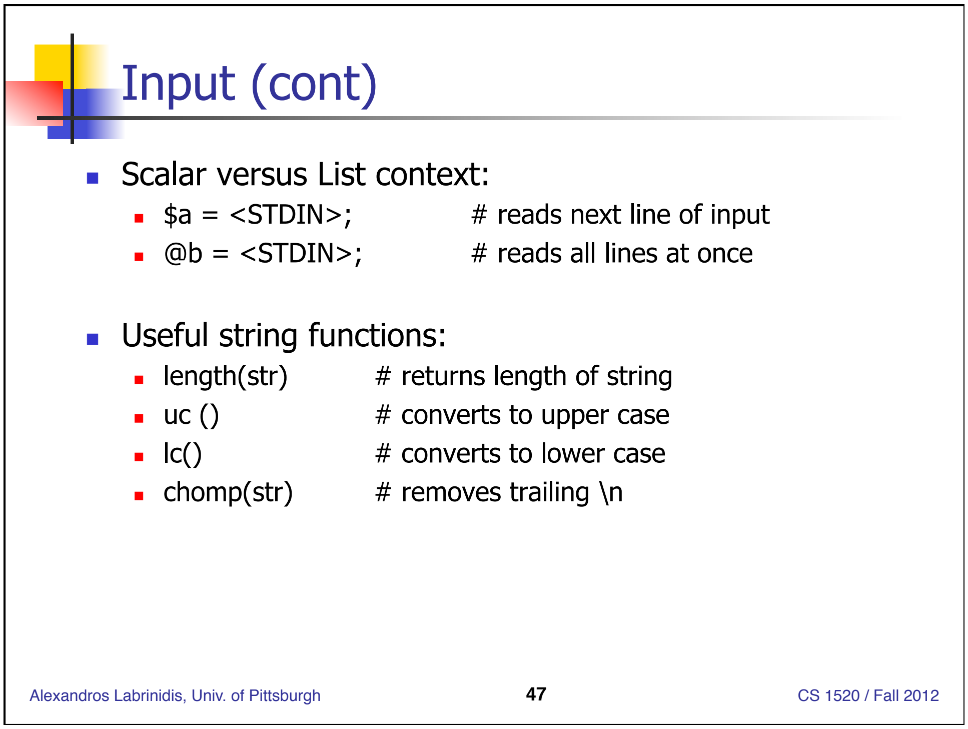


Input/Output

- Output:
  - **print** - simple
  - **printf** - formatted version

- Input:
  - standard input is accessed via <STDIN>
  - will return 0 if no more input
  - <> form is more general; can read from files
  - $_ to access line within loop

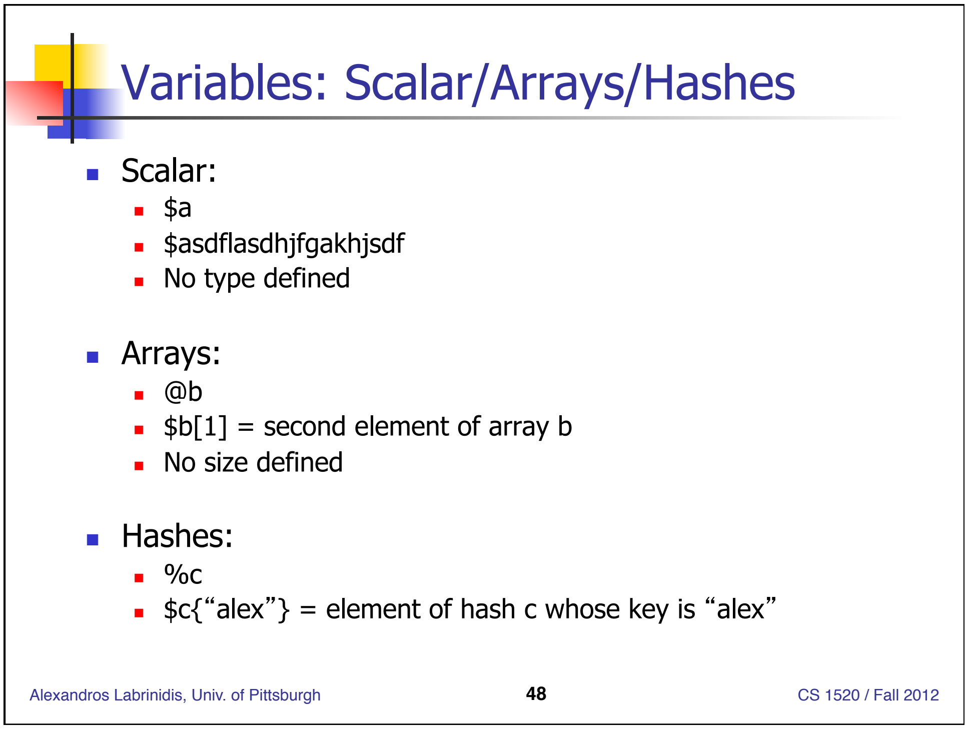
  -
    ```
    while (<STDIN>) {
      print $_;
    }
    ```



Input (cont)

- Scalar versus List context:
  - $a = <STDIN>;   # reads next line of input
  - @b = <STDIN>;   # reads all lines at once

- Useful string functions:
  - length(str)   # returns length of string
  - uc ()   # converts to upper case
  - lc()   # converts to lower case
  - chomp(str)   # removes trailing \n

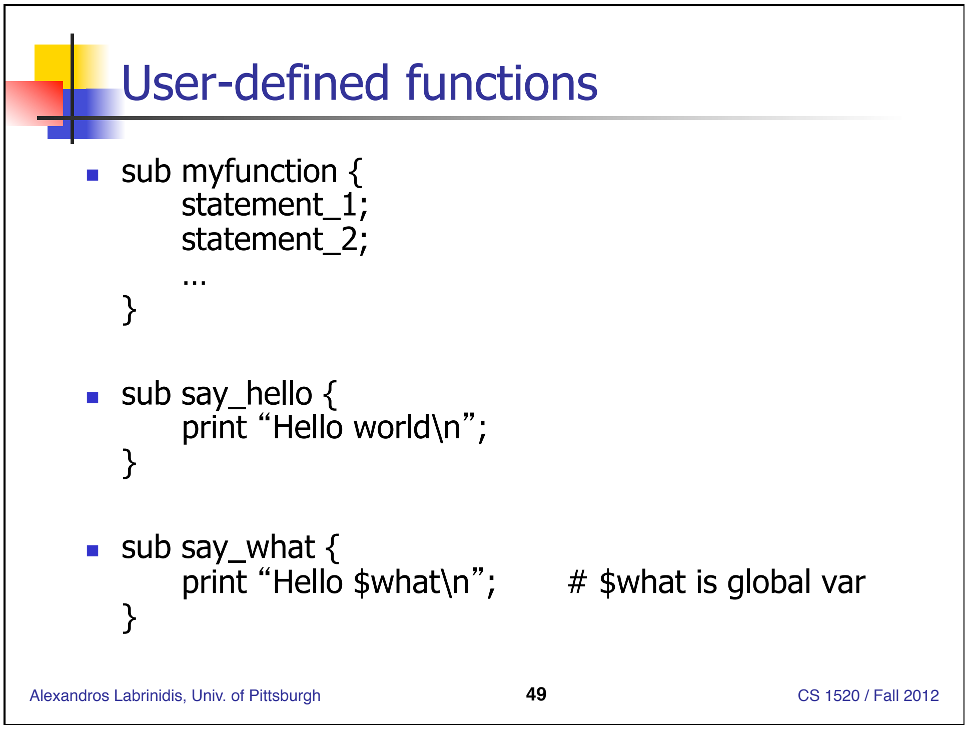


Variables: Scalar/Arrays/Hashes

- Scalar:
  - $a
  - $asdflasdhjfgakhjsdf
  - No type defined

- Arrays:
  - @b
  - $b[1] = second element of array b
  - No size defined

- Hashes:
  - %c
  - $c{“alex”} = element of hash c whose key is “alex”

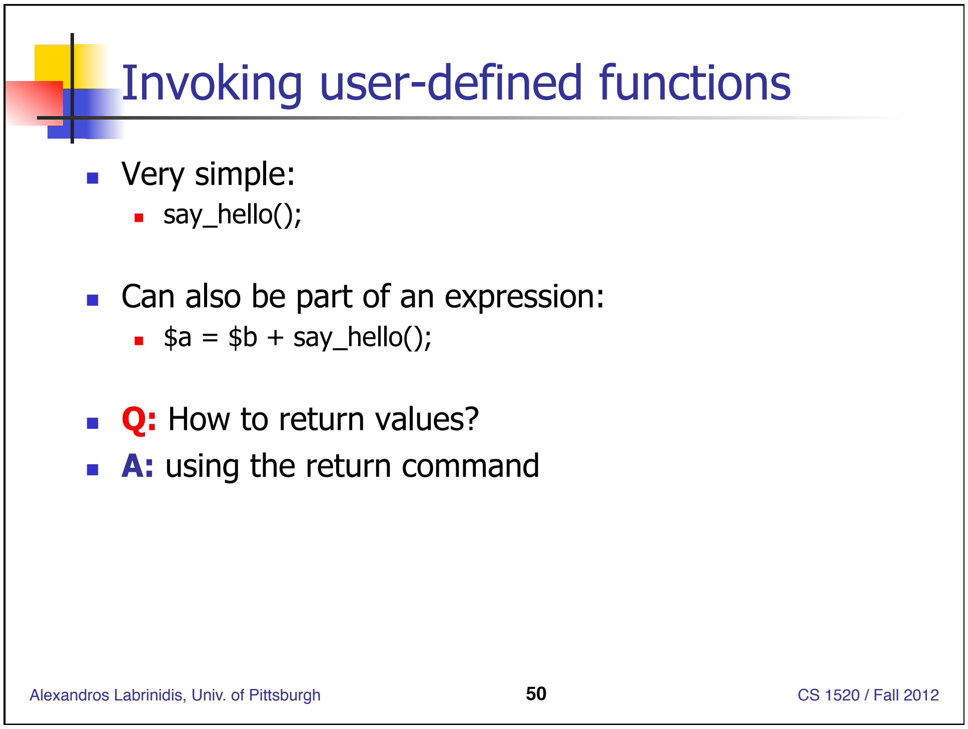


User-defined functions

-
  ```
  sub myfunction {
        statement_1;
        statement_2;
        …
  }
  ```

-
  ```
  sub say_hello {
        print “Hello world\n”;
  }
  ```

-
  ```
  sub say_what {
        print “Hello $what\n”;       # $what is global var
  }
  ```



Invoking user-defined functions

- Very simple:
  - say_hello();
- Can also be part of an expression:
  - $a = $b + say_hello();
- **Q:** How to return values?
- **A:** using the return command

How to pass arguments?

- @_ is a special array that holds all arguments to current function

- Two standard ways to use it:
  - $_[0] is the first argument, $_[1] is the second argument, …
  - ($a, $b) = @_;   #assigns first arg to $a and second arg to $b

- Examples:
  - addtwo.pl

Lexical Scope

- Global variables
  - Variables outside the function are accessible within the function
  - **Scope**: global

- Private variables in functions
  - my ($sum);
  - **Scope**: only valid while inside the current statement block (i.e., function)

- Semi-private variables in functions
  - local ($sum)
  - Scope: valid until function terminates
    (i.e., even if another function was called from within)

Examples

- spoof.pl

- add_any.pl:

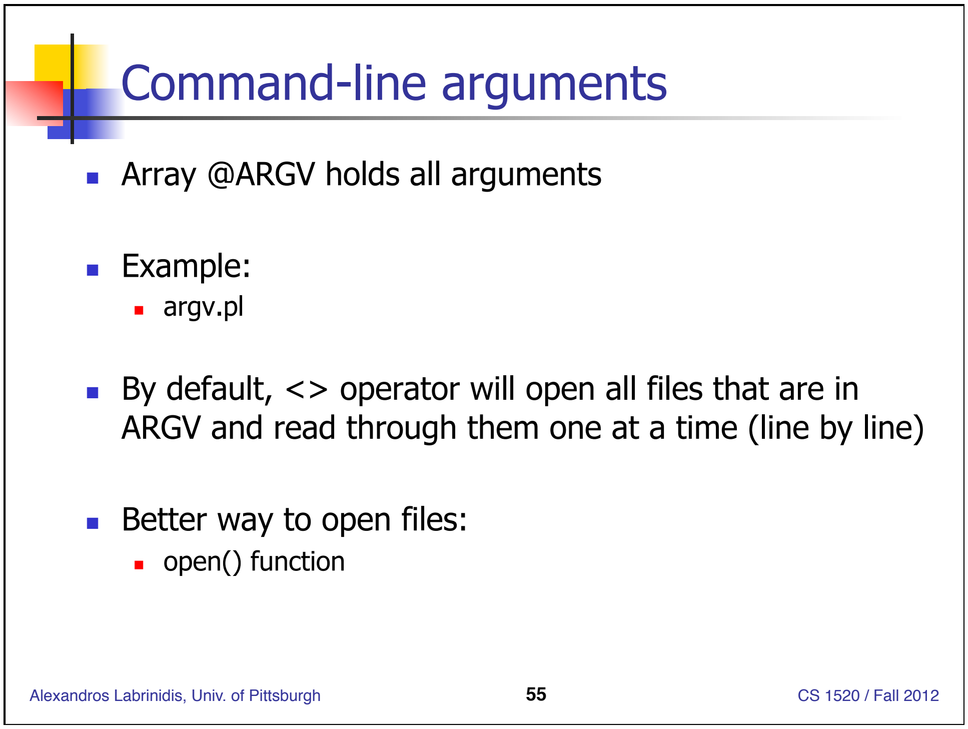
```
sub add {
    my ($sum);                 # private variable
    $sum = 0;                  # initialize
    foreach $param (@_) {      # go over all args
          $sum += $param;      # update sum
          }
    return $sum;               # return total
}
```

use strict;

- You can enforce "declaration" of variables by putting
  `use strict;`
  at the beginning of your program

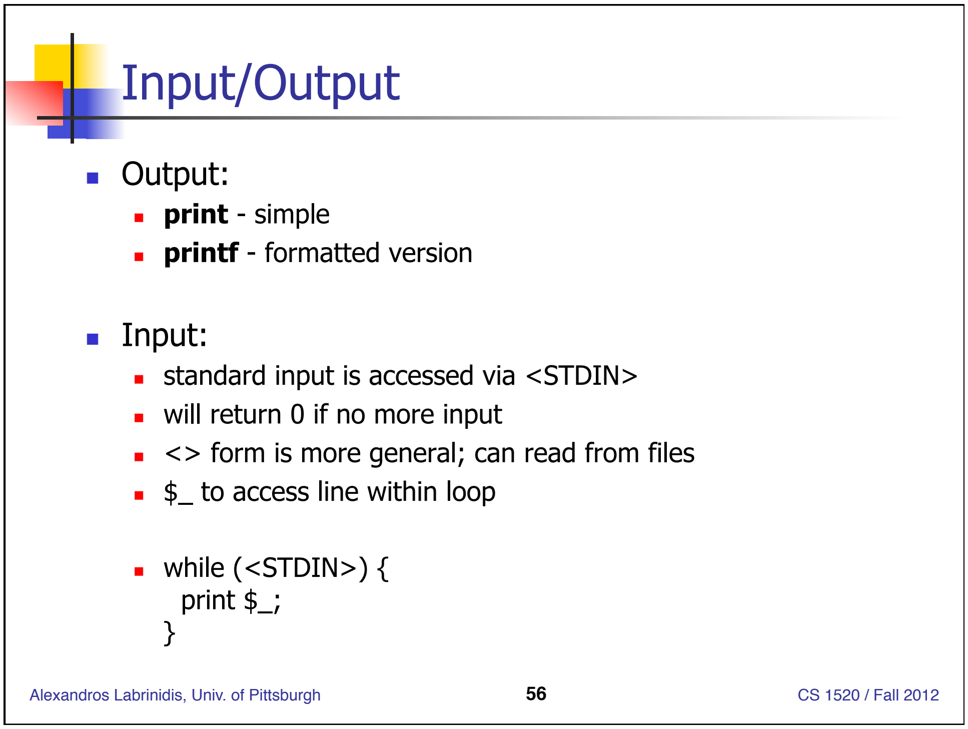
- Variables that are not "declared" will cause an error
  - Note: we are still only declaring variables by name. No type information (integer/string/etc) is given!

- Declare variables using my:
  - my ($a, $fred, $wilma);



Command-line arguments

- Array @ARGV holds all arguments
- Example:
  - argv.pl
- By default, <> operator will open all files that are in ARGV and read through them one at a time (line by line)
- Better way to open files:
  - open() function



Input/Output

- Output:
  - **print** - simple
  - **printf** - formatted version
- Input:
  - standard input is accessed via <STDIN>
  - will return 0 if no more input
  - <> form is more general; can read from files
  - $_ to access line within loop

  -
    ```
    while (<STDIN>) {
      print $_;
    }
    ```

Input (cont)

- Scalar versus List context:
  - $a = <STDIN>; # reads next line of input
  - @b = <STDIN>; # reads all lines at once

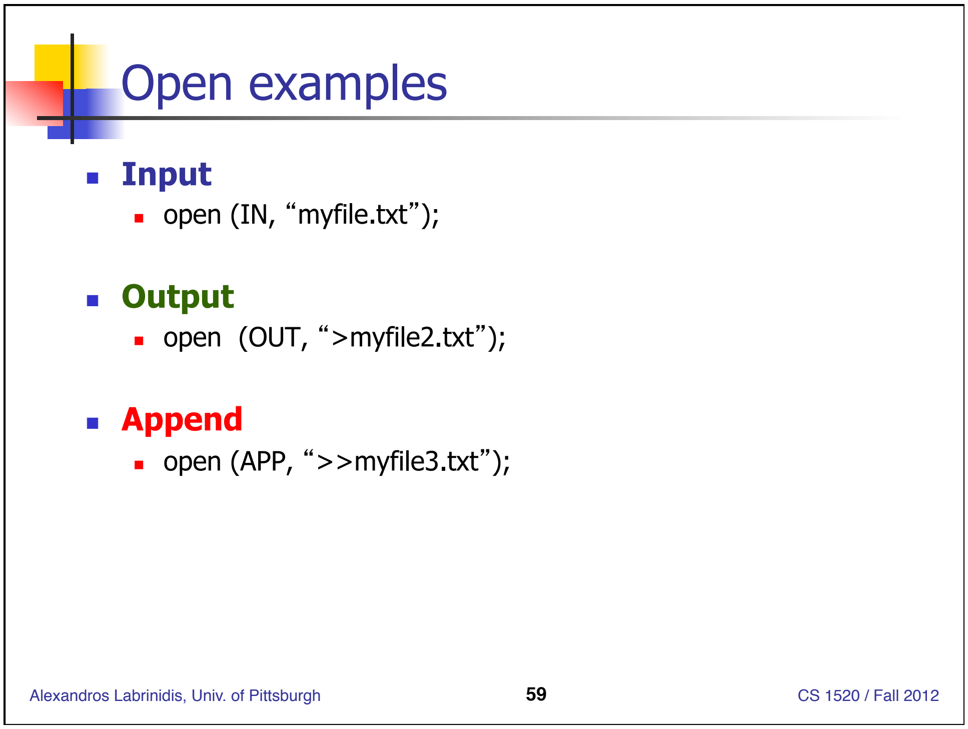
- Useful string functions:
  - length(str) # returns length of string
  - uc () # converts to upper case
  - lc() # converts to lower case
  - chomp(str) # removes trailing \n

File I/O

- **Filehandles**: name of an I/O connection
  - **`STDIN`**: standard input, from keyboard
  - **`STDOUT`**: standard output, your screen (where print goes)

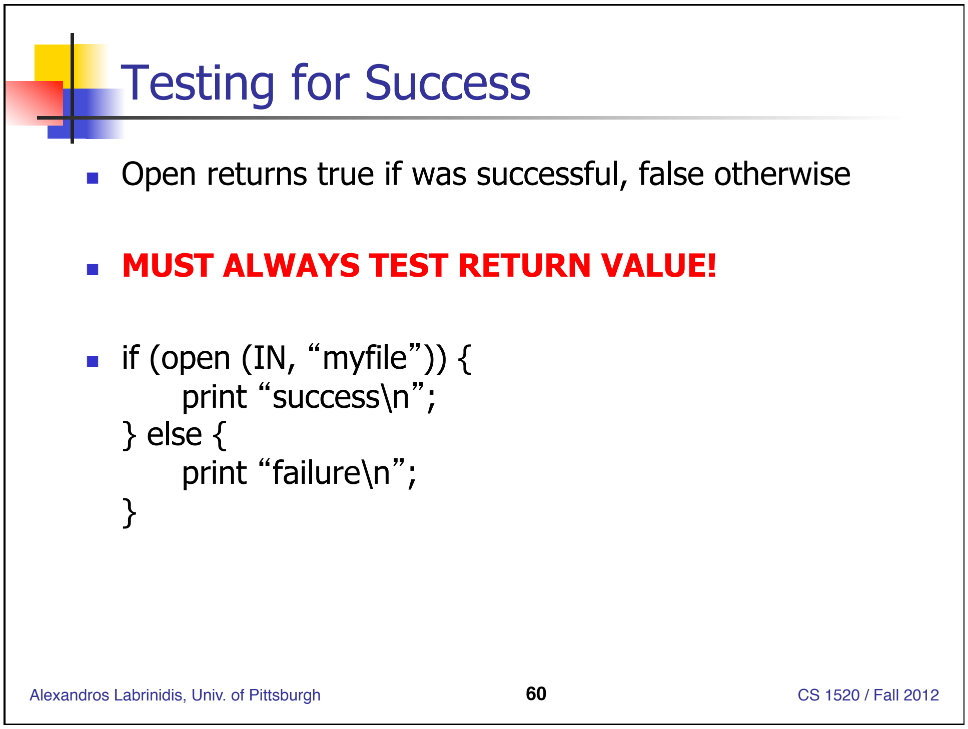
- open (FILE, “somename”);
  - FILE is the new filehandle
  - Somename is the external filename

- close (FILE);



Open examples

- **Input**
  - open (IN, “myfile.txt”);
- **Output**
  - open (OUT, “>myfile2.txt”);
- **Append**
  - open (APP, “>>myfile3.txt”);



Testing for Success

- Open returns true if was successful, false otherwise
- **MUST ALWAYS TEST RETURN VALUE!**

-
  ```
  if (open (IN, “myfile”)) {
        print “success\n”;
  } else {
        print “failure\n”;
  }
  ```

A simpler way

-
  ```
  if (open (FILE, “myfile.txt”)) {
        #success
  } else {
        #failure
        print “Sorry, I could not find myfile.txt\n”;
  }
  ```

- Perl’s shortcut: **die**()
  - Will print an error message and exit

- open (FILE, “myfile.txt”) || die (“Sorry, I could not find myfile.txt”);

File tests

- Test properties of files:
  - -s filename tests if file exists and has non-zero size

- Usage:
  ```
    if (-s $filename) {
        #file exists
  } else {
        # files does not exist
  }
  ```