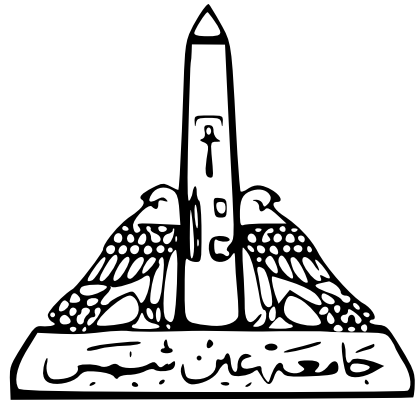


A Perl Crash Course

Session I

Joseph Samy

Ain Shams University

Feb. 2, 2013

About Me

- Joseph Samy Albert
- ECE ASU 2010.
- TA at ASU.
- Been using Perl for the past 3 years.
- **Not** a Perl expert.

About These Sessions

Setting Expectations

These sessions won't

- ✗ Teach you how to write scripts
- ✗ Teach you how to write Perl

These sessions will

- ✓ Teach you about scripting
- ✓ Showcase examples
- ✓ Start you on the road to Perl

About These Sessions

Setting Expectations

These sessions won't

- ✗ Teach you how to write scripts
- ✗ Teach you how to write Perl

These sessions will

- ✓ Teach you about scripting
- ✓ Showcase examples
- ✓ Start you on the road to Perl

About These Sessions

Organization

- **Not** lectures!
- Interact.
- Stop me any time.
- Experiment on your laptop!

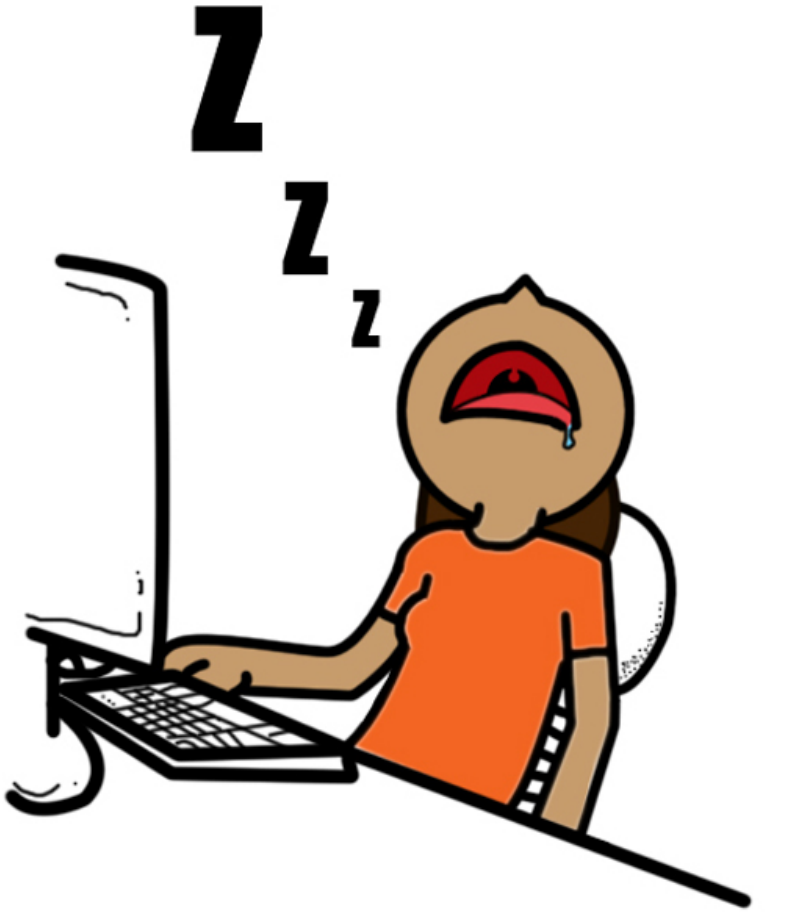
Introduction

Scripting
What?

- A scenario!
- Control your tools.
- **Automate**.
- Line-by-line.

Scripting
Why?

- Save time.
- Avoid error.
- Create a **workflow**.
- Achieve new things!



Scripting
When?

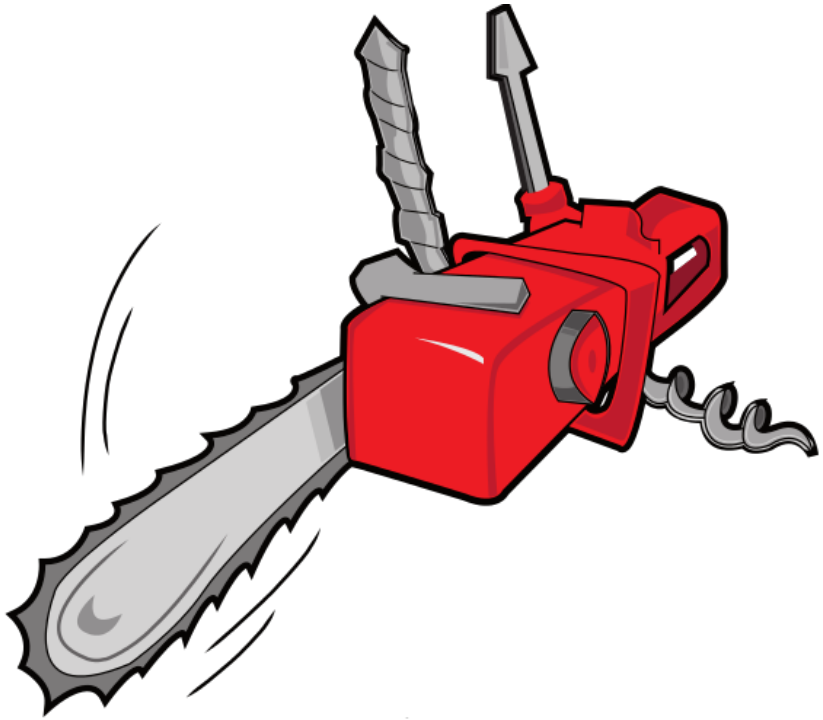
- Repetitive task
- Systematic task
- Error-prone task
- A quick modification to 1 file vs. the same modification in 100 files

Perl
Who?

- Larry Wall.
- Studied **natural and artificial languages**
- Developed Perl (1987) for report processing.
- Still a Perl developer!

Perl
Why?

- Cross-platform!!
- The best for **text manipulation**.
- The “Swiss Army Chainsaw”
- The **community**!!
- A personal choice . . .



Perl Culture

- The Perl slogans:

  There's more than one way to do it (TMTOWTDI)

  Easy things should be easy and hard things should be possible.

- Current version is 5.16
- Perl 6 in development for 12 years!

Perl Culture

- The Camel book (*Programming Perl*).
- Also, the Camel code.
- The onion is another symbol for Perl

Perl Culture

- The Camel book (*Programming Perl*).
- Also, the Camel code.
- The onion is another symbol for Perl

```
#!/usr/bin/perl -w                                        # camel code
use strict;

                                        $_='ev
                                    al("seek\040D
                 ATA,0,           0,");foreach(1..3)
              {<DATA>;}my       @camel1hump;my$camel-;
  my$Camel  ;while(          <DATA>){$_=sprintf("-%-6
9s",$_);my@dromedary      1=split(//);if(defin-ed($
_=<DATA>)){@camel1hu-m      p=split(//);}while(@-dromeda
ry1){my$camel1hump=0      ;my$CAMEL=3;if(defin-ed($_=shif
    t(@dromedary1    ))&&/\S/){$camel1hum-p+=1<<$CAMEL;}
   $CAMEL--;if(d   efined($_=shift(@dro-medary1))&&/\S/){
  $camel1hump+=1  <<$CAMEL;}$CAMEL--;i-f(defined($_=shift(
@camel1hump))&&/\S/){-$camel1hump+=1<<$CA-MEL;}$CAMEL--;if(
defined($_=shift(@ca-mel1hump))&&/\S/){$c-amel1hump+=1<<$CAME
L;;}$camel.=(split(/-/,"\040..m`{/J\047\1-34}L^7FX"))[$camel1h
ump];}$camel.="\n";}-@camel1hump=split(/\-n/,$camel);foreach(@
camel1hump){chomp;$C-amel=$_;y/LJF7\173\1-75`\047/\061\062\063-\
064\065\066\067\070/-;y/12345678/JL7F\175-\173\047`/;$_=revers-e;
 print"$_\040$Camel\n-";}foreach(@camel1hu-mp){chomp;$Camel=$_;-y
  /LJF7\173\175`\047/1-2345678/;y/12345678/-JL7F\175\173\0 47`/;
   $_=reverse;print"\04-05$_$Camel\n";}';s/\-s*//g;eval;   eval
    ("seek\040DATA,0,0;"-};undef$/;$_=<DATA>;-s/\s*//g;(    );;s
     /^*_:::;map{eval"pri-nt\"$_\"";}/.{4}/g; __DATA__   \124
      \1    50\145\040\165\163\1-45\040\157\1 46\040\1  41\0
           40\143\141  \155\145\1 54\040\1   51\155\  141
           \147\145\0  40\151\156 \040\141    \163\16 3\
            157\143\   151\141\16  4\151\1     57\156
            \040\167  \151\164\1   50\040\      120\1
            45\162\   154\040\15     1\163\     040\14
            1\040\1   64\162\1        41\144      \145\
            155\14    1\162\          153\04       0\157
             \146\    040\11      7\047\           122\1
             45\15    1\154\1   54\171             \040
             \046\       012\101\16                3\16
             3\15          7\143\15                1\14
              1\16           4\145\163              \054
             \040           \111\156\14             3\056
            \040\     125\163\145\14               4\040\
            167\1     51\164\1  50\0                40\160\
          145\162                                  \155\151
        \163\163                                    \151\1
      57\156\056
```

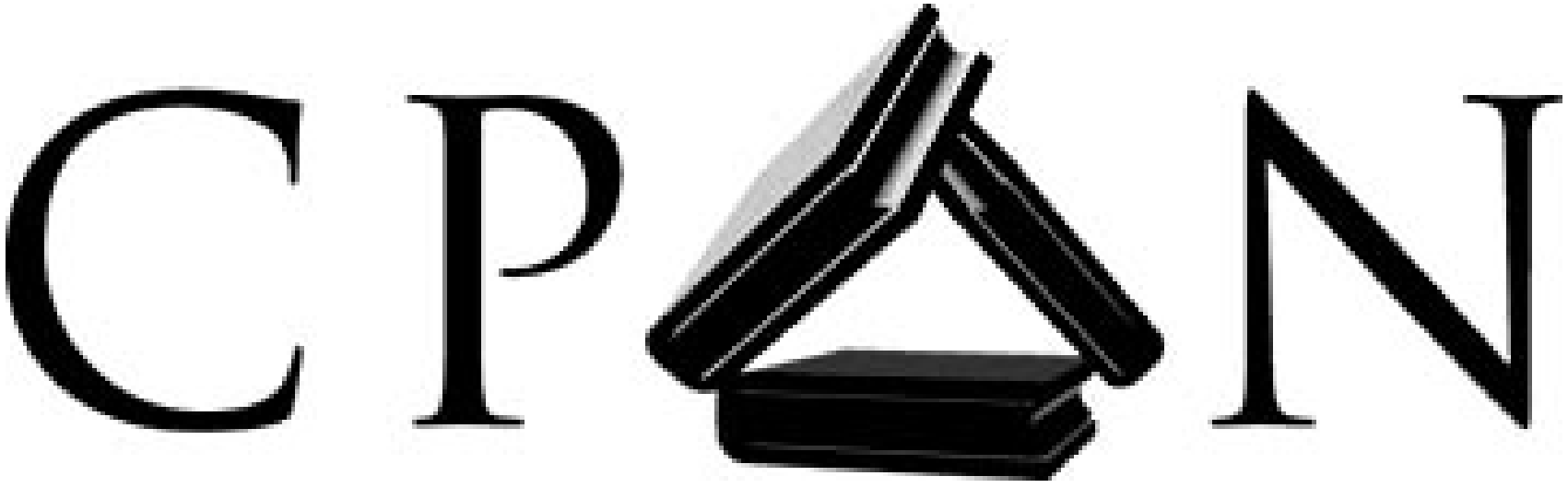
Perl Culture

- The Camel book (*Programming Perl*).
- Also, the Camel code.
- The onion is another symbol for Perl

Pearl Onion

Perl Onion

Perl Culture



- Comprehensive Perl Archive Network (www.cpan.org)
- Extensions to Perl (modules).
- No Perl without CPAN!
- More later...

Real Life Use Cases

- Bulk modification of netlists.
- Writing repetitive Verilog code.
- Automating PVT runs and post-processing.
- USB communication with lab instruments.
- Personalized bulk mail!

Session I

The Basics

Session I: The Basics

Hello, World!

The two main ways to run a Perl script (both equivalent on Windows):

1. Calling the Perl interpreter on a file:

Hello_World1.pl

```
1 print("Hello, world!\n");
```

```
[user@host]$ perl Hello_World1.pl
```

2. Putting the Perl "shebang" on the first line of the file:

Hello_World2.pl

```
1 #!/usr/bin/perl
2 print("Hello, world!\n");
```

```
[user@host]$ ./Hello_World2.pl
```

Hello, World!

The way you should begin **every** Perl script:

```
1 #!/usr/bin/perl
2
3 use strict; # The strict pragma
4 use warnings; # The warnings pragma
5
6 # In case you didn't notice, comments begin with a hash mark
      (#) and end at the end of the line.
```

Hello, World!

The way you should begin **every** Perl script:

```
1 #!/usr/bin/perl
2
3 use strict;
4 use warnings;
```

- The **use** statements are similar to the **#include** of C/C++. They are called pragmas.
- The **strict** pragma ensures that you can't use a variable without having declared it first.
- The **warnings** pragma makes Perl send you helpful warning messages when you mess things up.

Hello, World!

How `strict` Will Save Your Sanity

Imagine the following scenario:

```
1  #!/usr/bin/perl
2
3  use warnings;
4
5  my $abc = 2;
6  print("My scalar's value is ",$acb,"\n");
```

- The **warnings** pragma mercifully lets you know that you are trying to use an uninitialized value in a print statement.
- This is due to a simple typo. In a large script, this error can be very tricky to spot.
- As long as the `strict` pragma is not used, the undeclared variable ($acb) is OK to use as far as Perl is concerned.

Hello, World!

How `strict` Will Save Your Sanity

Now contrast it with the following:

myscript.pl

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  my $abc = 2;
7  print("My scalar's value is ",$acb,"\n");
```

- You now get a sane error saying:

```
[user@host]$ ./myscript.pl
Global symbol $acb requires explicit package name at
./myscript.pl line 7
```

Variable Types

- Much like the M language of MATLAB, Perl is dynamically-typed: you don't need to declare a variable is a float, int, char,...
- Instead, Perl defines three types of variables:

  Scalars

  Arrays

  Associative Arrays

Variable Types

- Much like the M language of MATLAB, Perl is dynamically-typed: you don't need to declare a variable is a float, int, char,...
- Instead, Perl defines three types of variables:

  Scalars Variables that contain a single value. This value can be an integer, a floating point number, a character, a string,... Scalar variables are denoted by a $ before their name.

  Arrays

  Associative Arrays

Variable Types

- Much like the M language of MATLAB, Perl is dynamically-typed: you don't need to declare a variable is a float, int, char,...
- Instead, Perl defines three types of variables:

  Scalars

  Arrays A collection of scalars. This means that one array can contain both numbers and characters. Array indexing in Perl is zero-based like C/C++. Array variables are denoted by an @ before their name.

  Associative Arrays

Variable Types

- Much like the M language of MATLAB, Perl is dynamically-typed: you don't need to declare a variable is a float, int, char,...
- Instead, Perl defines three types of variables:

  Scalars

  Arrays

  Associative Arrays More often referred to as **hashes**. They are like arrays but instead of using number indices, they use string indices. Think of them as dictionaries.

  Hashes are indicated by a % before their name.

Variable Types

Variable Declaration

Variables are declared using the **my** keyword:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  my $scalar;
7  my @array;
8  my %hash;
```

Using Variables

Using Scalars

```
1 my $a = 5; #Integer
2 my $c = 1.2e-5; #Float
3 my $b = 'b'; #Character
4 my $d = 'variable'; #Single-quoted string
5 my $e = "$d interpolation\n"; #Double-quoted string (variable
      interpolation)
```

- **Variable interpolation** means that referring to a variable inside a double-quoted string causes that variable's **value** to be inserted into the string.
- Double-quoted strings also allow backslash escapes similar to those of C/C++ (e.g. \n, \t,...)

Numbers and Strings

- Many number formats are accepted

```
1 23
2 1.2E-05 # scientific notation
3 5_000 # = 5000
4 0xA2 #hex
5 0b1000 #binary
```

- Perl is very flexible in the way it treats numbers.
- If you use mathematical operators to operate on number strings, Perl converts them to numbers automatically.
- Note the following:

```
1 print(1e0+"2"); #prints 3
2 print("2E-03" + '5e-3'); #prints 0.007
3 print(5_000_000 + 3_000_000); #prints 8000000 (5_000_000
      = 5,000,000)
```

Using Variables

Using Arrays

```
1  my $sc1 = 1.2e-5;
2  my @arr1 = (5 , 'b' , $sc1 , 'strings are OK too');
3  my @arr2 = (1 .. 10);
4  my @arr3 = ('a' .. 'x');
5  my @arr4 = (@arr3,'y' .. 'z'); #The alphabet
6  my @arr5 = qw(a b c Perl);
7  my $sc2 = $arr1[0]; #$sc2 now contains 5
8  my $sc3 = $arr3[-1]; #$sc3 now contains 'x';
9  my $sc4 = $arr3[-2]; #$sc3 now contains 'w';
10 my $sc5 = $#arr2; #$sc5 now contains 9
11
12 #Array Slices:
13
14 my @arr6 = @arr2[0,8]; #@arr4 contains (1,9)
15 my @arr7 = @arr2[2 .. $#arr2]; #@arr5 contains (3..10)
```

Using Variables

Using Arrays

- Perl defines four useful functions for dealing with arrays: push/pop (stack functions) and unshift/shift (queue functions).

```
1  my @a = ('a' .. 'f');
2
3  my $b = shift(@a); #$b == 'a' @a == ('b' .. 'f')
4
5  unshift(@a,'x'); #@a == ('x','b' .. 'f')
6
7  my $c = pop(@a); #$c == 'f' @a == ('x','b' .. 'e')
8
9  push(@a,'y'); #@a == ('x','b' .. 'e','y')
```

Using Variables

Using Hashes

```
1  my %person = (
2  name => "John Smith",
3  job => "Perl Developer",
4  age => 23
5  ); #No need to quote the keys
6  print($person{job},"\n");
7  my @fields = keys %person; #name, job,...
8  my @values = values %person; #John Smith,Perl Developer,...
9  if(exists $person{salary})
10 {
11    print("Salary found\n!");
12 }
13 my @person = ('name','John Smith','job','Perl Developer','age'
       ,23);
14 %person2 = @person; #Same person!
```

Using Variables

Using Hashes

A few notes on hashes:

- Hashes must have unique keys but not necessarily unique elements.
- The order in which hash elements are stored is unpredictable. In other words, keys will not return the keys in the same order you defined them.
- Hash access time is nearly independent of hash size: accessing an element in a 1000-element hash takes almost the same time as accessing one in a 10-element hash.

Using Variables

Stop and Think

- Can you think of applications for hashes?
- How can you get around the fact that hash element order is unpredictable?
- How can you take advantage of the fact that hash access time is nearly independent of hash size?

Operators

Arithmetic Operators

Operators	Comments
+ - / *	Simple arithmetic
**	Exponentiation (2**5 = 32)
%	Modulo (5%2 = 1)
++ −−	Increment/Decrement (contrast $c++ to ++$c)

Logical Operators

Operators	Comments
&& and	Logical AND
|| or	Logical OR
! not	Logical NOT

Operators

Relational Operators

Operators	Comments
== != > >= < <=	Numerical comparisons
eq ne gt ge lt le	String Comparisons

Bitwise Operators

Operators	Comments
~	Bitwise negation
| & ^	Bitwise OR, AND and XOR
<< >>	Bitwise shift left/right

Operators

Ternary Operator

Operators	Comments
?:	*<logical statement> ? <if true> : <if false>*

String Operators

Operators	Comments
.	String concatenation ('a'.'b' => 'ab')
x	String repetition operator ('a' x 3 => 'aaa')

Operators

Assignment Operators

Operators	Comments
=	Normal assignment
+= -= /= *= %= **=	Arithmetic assignment
~= >>= <<= &= |= ^=	Bitwise assignment
&&= ||=	Logical assignment
.= x=	String assignment

Example:

```
$x=3;
$x**=2;   #$x = 9
```

Operators
Short Circuit Operators

- It should be noted that Perl's logical operators short-circuit like those of C/C++
- Short circuiting is based on the fact that 1 && A = A while 0 && A = 0.
- Thus if the left operand to a && operator evaluates to false, Perl doesn't evaluate the right hand operand as there's no need for it.
- The same goes for || and and and or.
- Can you guess what the following statement does?

```
1 (defined $a) and (print $a)
```

Flow Control

Conditionals

The if statement syntax is similar to C/C++:

```
1 if( $a == 5 )
2 {
3    print $b;
4    #do something else;
5 }
```

If you have a single statement to execute after the conditional, use the more Perl-ish

```
1 print $b if( $a == 5 );
```

Flow Control

Conditionals

Need a negated logical condition? Use unless

```
1 unless( $a == 5 )
2 {
3    print $c;
4    #do something else;
5 }
```

A single-statement version also exists for unless

```
1 print $c unless( $a == 5 );
2 #Equivalent to print $c if($a != 5);
```

Flow Control

Loops

The same for, while and do−while you're familiar with

```
1  for(my $i=0;$i<10;$i++)
2  { }
3  ...
4  while($c < 5)
5  { }
6  ...
7  do
8  { }while($d > 2); #Note the semicolon
```

A much better (and more Perl-ish) way to write the above for loop:

```
1  for my $i (0..9)
2  { }
```

Flow Control

Loops

A negated version of while, called until, exists.

```
1 until($c >= 5)
2 { }
3 #Equivalent to
4 # while ($c < 5)
5 # { }
```

Perl also offers an additional (and very powerful) type of loop: the foreach loop. The foreach loop iterates over the elements of a list (array):

```
1 foreach my $letter ('a' .. 'z')
2 {
3   print $letter,"\n";
4 }
5 #Prints the alphabet
```

Flow Control

Loops

foreach can also iterate over an array variable:

foreach_test.pl

```
1 my @a = ('Perl is ','a great ','language!',"\n");
2 #Note the double-quoted string for the newline character!
3 foreach my $element (@a)
4 {
5   print $element
6 }
```

```
[user@host]$  ./foreach_test.pl
Perl is a great language!
[user@host]$
```

Flow Control

Loops

Example: printing the contents of a hash

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5
6  my %person = ( name => 'John', job => 'Perl Developer',
7                 salary => 'Big Bucks');
8  foreach my $key (keys %person)
9  {
10   print("$key\t$person{$key}\n");
11   #Or:
12   #print($key,"\t",$person{$key},"\n");
13 }
```

Flow Control

Loops

- In C/C++, interrupting the normal flow of a loop was done using break and continue; these are called last and next, respectively, in Perl.
- The following example uses an **imaginary** function called `elements` that returns a list of elements of a given string (whether they're words or spaces).
- The example assumes we want to count the number of times each word has occurred in a given piece of text.
- After counting the occurrences, the program will check if certain words exist in the text and if so, print the number of times each word occurred.

Flow Control

Loops

```
1  #Suppose the text is in the variable $string
2  foreach my $element (elements $string)
3  {
4     last if($element eq 'EOF'); #End of file
5     next if($element eq ' '); #Don't count spaces!
6     exists $count_words{$element} ?
7     $count_words{$element}++ : $count_words{$element}=1;
8  }
9
10 my @words = ('word1','word2','word3');
11 foreach my $word (@words)
12 {
13    next unless(exists $count_words{$word});
14    print("$word occured $count_words{$word} times\n");
15 }
```

Subroutines

Creating Subroutines

- Subroutines are similar to C/C++ **functions**.
- Unlike C/C++ functions, they **don't** need to be declared before they're called and **don't** (necessarily) have a prototype.
- Perl's subroutines are **variadic**, which means they can accept an arbitrary number of arguments.
- Subroutine arguments are stored in the **special variable** @_
- Syntax for creating a subroutine:

```
1  sub mysub
2  {
3      my $arg1 = $_[0];
4      my $arg2 = $_[1];
5      my @rest = @_[2 .. $#_];
6      # Do something;
7      return $result; #Or return @results;
8  }
```

Subroutines
Creating Subroutines

- The more conventional way to do it:

```
1  sub mysub
2  {
3      my $arg1 = shift(@_);
4      my $arg2 = shift(@_);
5      my @rest = @_; #The shifts have removed the first two
           arguments
6      #Do things
7      @results; #or $result
8  }
```

- Note that the value of the last evaluated statement is automatically returned.
- Use return when you want to return a value from inside the function body instead of the last statement.

Subroutines

Calling Subroutines

- Think of subroutines as the fourth data type.
- You specify that you want to call a subroutine by prepending an ampersand (&) to the subroutine name.

```
1 my @results = &mysub(@args);
2 my @results2 = &mysub($arg1,@rest);
3 #And so on ...
```

Subroutines

You Try

- Write a Perl script that would calculate the prime factorization of 1234567
- Write a Perl script that implements one of the hash applications you thought of.

User Input

- You receive input from the user via the `STDIN` file handle.
- Assume the following example:

user_in.pl

```
1 print("Please enter your name: ");
2 $user_name = <STDIN>;
3 print("Your name is John\n") if ($user_name eq 'John');
```

```
[user@host]$  ./user_in.pl
Please enter your name:  John
[user@host]$
```

- Can you figure out what happened?

User Input

- The user entered his name and then pressed ENTER. By default, Perl doesn't remove this character and $user_name will contain "John\n", making the comparison fail.
- To deal with this, Perl has a function called chomp that removes the newline character from the end of its argument:

user_in.pl

```
1 print("Please enter your name: ");
2 $user_name = <STDIN>;
3 chomp($user_name);
4 print("Your name is John\n") if ($user_name eq 'John');
```

```
[user@host]$  ./user_in.pl
Please enter your name:  John
Your name is John
[user@host]$
```

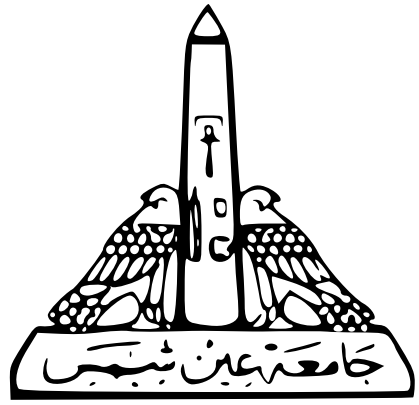
Homework

- This homework is **not** for me!

Homework

- Think of ways you can use Perl at work/college/home.
- Visit the CPAN and search for modules related to your area(s) fo interest.
- Modify your prime factorization script to prompt the user for the number of times they want the program to run. For that many times, prompt the user for a number and print out its prime factorization. What happens if the user enters a non-numeric input?

End of Session I.
You haven't learned any Perl yet!!



A Perl Crash Course

Session II

Joseph Samy

Ain Shams University

Feb. 4, 2013

Session II

The Useful Stuff

Session II: The Useful Stuff

Context

- Context refers to the data type **expected** from an operation.
- Very important concept since a lot of Perl functions change their behavior depending on the context.
- There are only two contexts in Perl: **scalar** context and **list** context.
- Perl determines context based on operators, the variable being assigned to and certain defaults.
- Subroutine arguments are automatically placed in list context, which is why the following statements are equivalent:

```
1 my @args = ($one,$two,$three);
2
3 &my_sub($one,$two,$three);
4 &my_sub(@args);
```

Context

```
1  my @arr = ('one','two','three');
2
3  #Array evaluated in scalar context:
4  my $element_count = @arr; # $element_count = 3
5  my $result = 3 + @arr; # $result = 6
6
7  print(@arr); #print evaluates its arguments in list context
8  #prints "onetwothree"
9
10 #Forcing scalar context:
11 print(scalar(@arr)); #prints 3
```

Regular Expressions

Introduction

- This is where Perl really shines.
- Regular expressions are probably the best known tool for string manipulation. (The "mathematics of strings")
- Regexes will probably make the main part of most of your scripts.
- Perl supports regex **natively**: no need for extra modules or classes.
- Regular expressions are a **vast** topic. This is only an introduction.
- For more on the subject, refer to the book *Mastering Regular Expressions*.

Regular Expressions

Introduction

- Regexes are a language to describe parts of a string.
- Used for search (and replace) operations.
- The simplest regular expression is a string.
- Regular expressions can grow much more complicated than simple strings. To do this, several special characters (called **metacharacters**) exist to modify the meaning of a regex.
- For example the period character (`.`) matches any non-newline character.
- To specify a literal period, escape it with a backslash in your pattern (most metacharacters can be escaped this way).

Regular Expressions
The Binding Operators

- Perl uses a special operator to indicate that a certain string should be matched against a regex, called a **binding operator**:

```
1 my $string = 'abc123def';
2 $string =~  /123/; #Not useful
3
4 my $matches = $string =~  m/abc/; #The m is optional
5 print("Matches!\n") if($string =~  /abc/);
```

- There's another binding operator which returns true when the string doesn't match the regex:

```
1 print("Doesn't match!\n") if($string !~ m/abc/);
```

Regular Expressions

The Binding Operators

- Using m before the string pattern allows yo to change the delimiters:

```
1 m!pattern!
2 m{pattern}
3 m(pattern)
4 m|pattern|
```

Regular Expressions

Anchors

- Would the regex /hat/ match the string "That cat."?
- It would. The regex is not the **word** "hat"; rather the letter 'h' followed by 'a' then 't'.
- Which part of "That hat." would /hat/ match?
- We need to be able to specify that we want the pattern match to occur **at a certain place** within the string. That's where anchors come in.

Regular Expressions

Anchors

- Anchors are **zero-width assertions**: they are parts of the regex that don't match any characters. They rather match a **place** within the string.
- The most important anchors are
  - ^ Matches the beginning of the string
  - $ Matches the end of the string
  - \b Matches a word boundary (i.e., the place between a non-word character and a word-character).
- Thus /foo/ would match 'foobar' and 'barfoo' but /^foo/ would only match 'foobar'.
- The regex /eat/ would match the words 'eat', 'eating', 'great', 'treatment',.... However, the regex /\beat/ would only match words starting with 'eat' while /eat\b/ would match only words ending with 'eat'.

Regular Expressions
Quantifiers

- Quantifiers tell Perl how many times to match the element before them.
- These are the quantifiers Perl recognizes:

Quantifier	**Meaning**
*	Match any number of times, including zero
+	Match any number of times, excluding zero (i.e. at least once)
?	Match one or none (i.e. an optional part of the pattern)
{n,m}	Match at least n times and at most m times. For example ? is equivalent to {0,1}

Regular Expressions
Quantifiers

- For example, the regex /a.*t/ would match 'at', 'act', 'appointment',...
- On the other hand /a.+t/ wouldn't match 'at', while /a.?t/ would match 'at' and all three-letter words starting with an 'a' and ending with a 't'.
- Furthermore, /a.{2,3}t/ would match all words starting with an 'a', ending with a 't' and containing 2 to 3 more characters in between.
- Test yourself: would /a.{0,5}t/ match against 'armament', 'atonement', 'management'?

Regular Expressions
Alternation

- Alternation means having several alternatives to match against: if one fails, try the next.
- So /cat|act/ would match 'cat', 'catering', 'act', 'acting', 'facts',...
- You can have more than two alternatives: /cat|act|a.*m/
- Note that Perl reads the alternatives from let to right and stops at the first match it finds.
- Thus, /foo|foobar/ is redundant, because the `foobar` part is never reached. The correct way to write it is /foobar|foo/. The rule is: write the shortest alternative last.

Regular Expressions

Character Classes

- Character classes are a way of referring to a whole group of characters at once.
- You enclose a character class between square brackets.
- The regex /[abc]/ matches **only one** character: an 'a', a 'b' or a 'c'.
- You can also define ranges of characters: [a−zA−Z] (any letter of the alphabet) [0−9] (any digit).
- To define a 'minus' character, either put it at the start of the class or escape it with a backslash. Both regexes /[−+]/ and /[+\−]/ work to specify 'plus or minus'. Why didn't we need to escape the '+'?
- A character class is negated by adding a caret as the first character within it: /[^0−9]/ would match a non-digit character.

Regular Expressions

Character Classes

- Some shortcuts exist to specify meaningful character classes:

Shortcut	Meaning
\w	A word character: any letter, digit or underscore. In short, it matches a valid variable name.
\W	The opposite of \w
\d	Any digit
\D	The opposite of \d
\s	Any whitespace character
\S	The opposite of \s

Regular Expressions
Grouping

- Sometimes you want to group part of the regex together so a quantifier can apply to more characters or so that alternation is limited to certain parts of the regex.
- Examples:

```
1 my $regex = 'foo(bar)+'; #The + applies to (bar)
2 $string =~ /$regex/;
3
4 my $regex2 = 'foo(bar|baz)'; #Limited alternation
```

- The best thing about grouping parentheses is that they act as memory, storing what the part in parentheses has matched.

Regular Expressions
Grouping

- The special variables $<digit> store the contents of the parentheses. For example:

```
1 if('123.456' =~ m/(\d+)\.(\d+)/)
2 {
3   print("Integer part is $1 and fractional part is $2\n");
4 }
```

- In case of nested groups, the groups are numbered according to which parenthesis opened first.

```
1 my $string = 'eng.joseph.riad@gmail.com';
2 if($string =~ /(([\w.]+)@([\w.]+))/)
3 {
4   print("My address is $1, my name is $2 and the domain is
        $3\n");
5 }
```

Regular Expressions
Grouping

- What if we want to match numbers whose fractional part is the same as their integer part?
- To refer to the contents of a matched group **inside** the pattern, use a backslash instead of the $ prefix to the match variable:

```
1 my $string = '123.456';
2 if($string =~ /(\d+)\.\1/)
3 {
4   print("Match\n");
5 }
```

- This is called a **backreference**.

Regular Expressions
Grouping

- Note that the match variables will store the value of the **last successful match**. This means that if your current regex doesn't match, the values of the match variables may be from a different regex matched against a different string. This is why it's good practice to put your regex match in an if statement before working with the match variables:

```
1  '123' =~ /(\d+)/;
2  'abc' =~ /(\d+)/;
3
4  print("$1\n"); #Prints '123';
5
6  #Better:
7  print("$1\n") if('abc' =~ m/(\d+)/);
```

Regular Expressions
Lazy Quantifiers

- You should note that quantifiers by default are **greedy**. This means they will match as many characters of your string as they can. This can be a problem. Imagine the following scenario:

```
1 my $string = '(text1) and (text2)';
2 if($string =~ /\((.+)\).*/)
3 {
4   #Do things
5 }
```

- This will not work out as you expect. The .+ part would match everything before the **second** closing parenthesis because the quantifier + is greedy.
- The solution is to use a **lazy** quantifier: a quantifier that only matches characters if it has to. To do this, add ? to your quantifier.

Regular Expressions

Lazy Quantifiers

- Better ways to write the above regex are /\((.+?)\).*/, /\((\w+)\).*/ or even /\(([^)]+)\).*/
- The lazy quantifiers are +?, *?, ?? and {n,m}?

Regular Expressions
Modifiers

- Modifiers are single letters added after the closing / of the regex and modify its behavior.
- The most important ones are /i, /s, /m and /g.
- The /i modifier means 'ignore case'. Thus, an 'a' in the pattern would match either an 'a' or an 'A'.
- The /s modifier allows the . metacharacter to match newline characters. Use it when you want . to match across several lines.
- The /m modifier makes ^ match just after a newline and $match just before a newline. You can thus use these anchors when working with a slurped file for example (see later), to match specific lines. In this case \A would refer to the very beginning of the string and \Z to the very end.

Regular Expressions
Modifiers

- The /g modifier will apply the regex several times to the same string. Use it if you expect to match a regex several times:

```
1 my $string = '123 456 78';
2 while($string =~ /(\d+)\b/g)
3 {
4    push @numbers,$1;
5 }
```

- The /g modifier returns all the matched groups when in list context:

```
1 my @matches = $string=~/(patt1) and (patt2)/g;
```

- Modifiers can be combined and their order is unimportant.

Regular Expressions
Substitution

- So far, we've only seen how to search for a pattern inside a string using the m/<pattern>/ operator.
- There's also a substitution operator s/<pattern>/<replacement>/ for search and replace.
- It admits the same syntax rules and modifiers as the normal matching operator.
- Examples:

```
1 $string =~ s/[()]//g; #Delete all parentheses
2
3 $string =~ s/(\w+)\s+\1/$1/gi; #What does this do?
```

Regular Expressions

You Try

- Use regex to validate user input for the prime factorization script from last time.
- Write a regex to match numbers in scientific notation (e.g. 1e06 or -1.38E-23).

FILE I/O

- To really tap into the power of Perl, you need to learn to open text files.
- In Perl, you can open a file in one of three modes:
  1. Read
  2. Write
  3. Append (like write but without erasing existing file contents)
- Perl isn't limited to text files, though. If you have the right knowledge, you can edit binary files. Some people use Perl for graphics processing, for example.
- To open a file, you give Perl three pieces of information:
  1. A **filehandle**. This is like a special variable you use to read lines from a file.
  2. The **mode** you want to open the file in.
  3. The path to your file (absolute or relative to the current working directory).

FILE I/O

```
1  open(INFILE,'<','my_text_file.txt'); #Read mode
2  open(OUTFILE,'>','my_text_file.txt'); #Write mode
3  open(AOUTFILE,'>>','my_text_file.txt'); #Append mode
4  #Note that filehandle names are usually in all caps.
5  #Or:
6  open(my $fh,'<','my_text_file.txt'); #$fh can't be already defined
7
8  #Do some stuff
9
10 close(INFILE);
11 close(OUTFILE);
12 close(AOUTFILE);
13 close($fh);
14 #Always better to close your filehandles when you don't need
       them
```

FILE I/O

```
1  open(INFILE,'<','my_text_file.txt'); #Read mode
2  my $line1 = <INFILE>; #Read one line with diamond operator
3  my $line2 = <INFILE>; #and another one.
4
5  #Line-by-line processing:
6  while(defined(my $line = <INFILE>))
7  {
8     #Do stuff to $line
9  }
10 #while(my $line=<INFILE>) also works
```

- Note that if you had used the diamond operator twice before the while loop as shown above, the loop will start reading the file starting from the **third** line!

FILE I/O

```
1 open(OUTFILE,'>','my_text_file.txt'); #Write mode
2 my $string = "Line to print to file\n";
3 print(OUTFILE $string); #No comma!!
4 print(OUTFILE "What is the number of this line?");
5
6 #You can also use printf if you like it:
7 printf(OUTFILE "Interpolating a variable: %s",$string); #Note
      the commas!!
```

- Be careful when printing lines to a file opened in write mode!
- The first print you do will erase the whole file, with every subsequent print appending to the file.

FILE I/O

Staying Safe

- It's generally a disaster if your script keeps running after a failed file open. Every statement that processes the opened file will fail!
- The "standard" way to open files in Perl:

```
1 my $input_file = 'in.txt';
2 open(MYFILE,'<',$input_file)
3       or die("Couldn't open $input_file\n");
```

- Also, if you're feeling extra paranoid:

```
1 close(MYFILE) or die("Failed to close $input_file\n");
```

FILE I/O

You Try

- Create a dummy text file with random text then write a script to print it out in reverse into another file.
- Write a script that prints out a word count for your text file.

System Calls

- A system call means that you send a command directly to the OS.
- In Linux, system calls are mainly shell commands; in Windows, they're DOS commands.
- There are three ways to do a system call in Perl:
  1. Using the system function. This simply passes its arguments to the OS and waits for execution to terminate. Its return value is the same as the return value of the OS after commands have executed.
  2. Using back-ticks (`` `<commands>` ``). This is the same as system but the return value is the output of the command instead.
  3. Using the exec function. This function passes its arguments to the OS and exits. No code in the script is executed after an exec. If you use warnings, you'll get a warning if you have written code after an exec line.

System Calls

Linux examples:

```
1 system("echo \"Hello, world\"");  #Why a backslash?
2
3 #The back-tick button is to the left of the number '1'
4 my @files = `ls /tmp`;
5
6 exec("\$HOME/scripts/other_script.pl");  #Why a backslash?
```

- One tricky thing about Linux system calls is that (most) Linux shell commands return zero on success and a nonzero value on failure.
- A safe system call:

```
1 system("mkdir /tmp")
2         and die("$0 Couldn't run the command: $!\n")
```

Special Variables

- Perl has a **lot** of special variables that are used to perform special operations, change the behavior of Perl and obtain information about your Perl session.
- You can find the complete list of Perl special variables by running `perldoc perlvar` from the terminal.
- It makes sense to memorize some of them and what they represent.

Special Variables

Variable	Meaning
`@ARGV`	Array containing the command line arguments you pass to the script when calling it. Example: `./mysrcipt.pl arg1 arg2`
`$_`	The default argument for most functions (like print and chomp), the default iterator of for and foreach and the default placeholder for the diamond operator.
`@_`	The default array argument for the operators shift and pop
`%ENV`	A hash containing the environment variables defined in the shell you used to start the script. For example `$ENV{HOME}` returns the path to your home folder.

Special Variables

Variable	Meaning
$/	Input record separator. This is the character that the diamond operator uses to know where a line ends. By default this is "\n". Assign undef to it, to **slurp** a complete file into one string.
$^O	Returns the operating system name so you can write portable scripts!
$0	The name of your script
$!	The last error reported by the OS

Special Variables

Examples of the usage of $_:

```
1  #Default placeholder for diamond operator,
2  # default argument for print and default regex match target:
3  /^(abc)/ and print while(<FILE>);
4  #Default loop iterator:
5  print "$_\n" foreach(@array);
6
7  #Abuse:
8  for (0..9)
9  {
10   for(0..9)
11   {
12     #What does $_ refer to here?
13   }
14   # and here?
15 }
```

Special Variables

Can you figure out the purpose of the following subroutine?

```
1 sub x
2 {
3    my $y=@_;
4    unshift(@_,(shift)+(shift)) while($#_);
5    (shift)/$y;
6 }
```

Special Variables

Can you figure out the purpose of the following subroutine?

```
1  sub x
2  {
3     my $y=@_;
4     unshift(@_,(shift)+(shift)) while($#_);
5     (shift)/$y;
6  }
```

Answer:

- First, the number of input arguments is stored in $y(array in scalar context).
- The while loop will stop when there's only one element remaining in @_ (that is, when $#_ == 0)

Special Variables

Can you figure out the purpose of the following subroutine?

```
1  sub x
2  {
3     my $y=@_;
4     unshift(@_,(shift)+(shift)) while($#_);
5     (shift)/$y;
6  }
```

Answer:

- The loop shifts two elements from @_ and adds them, unshifting the result back onto @_.
- By the end of the loop, @_ should contain the sum of the arguments.

Special Variables

Can you figure out the purpose of the following subroutine?

```
1 sub x
2 {
3    my $y=@_;
4    unshift(@_,(shift)+(shift)) while($#_);
5    (shift)/$y;
6 }
```

Answer:

- The final statement returns the sum of the arguments divided by their number: the **average**.

Special Variables

A better way of opening files:

```
1 open(MYFILE,'<',$file)
2     or die("$0 : failed to open $file for reading: $!\n")
```

- This way, if your script was called by another script or program, it would be obvious which script caused the error.
- The error reported by the OS can sometimes be helpful for debugging.

Special Variables

File Slurping

- Sometimes it's better to process a file as a whole rather than line-by-line.
- Reading the whole file into a single string is called **slurping** the file.
- This is done by undefining the record separator variable.

```
1 open(FH,'<',$file)
2     or die("$0 : Couldn't open $file for reading: $!\n");
3 my $rec_sep = $/;
4 $/ = undef; #or undef $/;
5 my $slurp = <FH>; #The whole file appears as a single line to
      <>
6 close(FH);
7 $/ = $rec_sep;
```

Special Variables

File Slurping

Stop and Think

- Can you think of a scenario where it's better to slurp a file rather than read it line by line?

Useful Functions

Here are some useful Perl functions you should be aware of:

- `chdir` and `mkdir`
- `rename`
- `glob`
- `split`
- `join`

Useful Functions

chdir and mkdir

- These functions are similar to Bash's `cd` and `mkdir`, respectively.
- The main difference is that these functions are **portable**: they will work on any OS that Perl works on as opposed to a system call.
- Note that mkdir will create any intermediate directories as necessary.

```
1 chdir("/tmp") or die("Failed\n");
2 mkdir("/tmp/log/perl/scripts") or die("Failed\n");
```

- On Windows, paths would be written with a backslash instead.
- For a portable way of writing paths, look at the `File::Spec` module.

Useful Functions

rename

- This is similar to the `mv` command in Bash, but it's also portable.
- The rename command is a useful place to learn about the **fat comma** operator.
- We saw this operator before (=>) in hash declaration. In Perl grammar, this operator is syntactically equivalent to a comma, it just gives a more readable appearance to the function call:

```
1 rename($source=> $destination);
2 #Or:
3 rename($source,$destination);
```

Useful Functions

glob

- This is a portable way of globbing many files from the file system.

```
1 my @code_files = glob("*.pl *.c *.v");
2 #Or:
3 $_ = "*.pl *.c *.v";
4 my @code_files = glob;
```

Useful Functions

Useful Functions

- This function takes two arguments: a regex and a string.
- It returns an array whose elements are formed by splitting the string according to the regex
- Examples:

```
1 my $string = "aBc,Def , gHi";
2 my @a = split(/,/,$string); #Same as:
3 @a = ('aBc','Def ',' gHi')
4 my @b = split(/\s*,\s*/,$string); #Same as:
5 @b = ('aBc','Def','gHi')
6 my @c = split(/[A-Z]/,$string); #Same as:
7 @c = ('a','c,','ef , g','i')
```

Useful Functions

join

- This is the opposite of split. It takes a string and an array as input.
- It returns a string consisting of the elements of the array joined together by the given string (**not** regex).
- Example:

```
1 my @a = ('one','two','three');
2 my $string = join(',',@a); #Same as:
3 $string = 'one,two,three';
```

Extending Perl: the CPAN

- As stated before, the CPAN is almost an essential part of Perl.
- If you want to perform a specific task using Perl, you should first search the CPAN to make sure it hasn't already been done.
- Check the sites www.cpan.org and metacpan.org.
- Instead of downloading the module from the CPAN, you can also run the following in a terminal:

```
[user@host]$ cpan install <module>
```

This is better than using the website for download as it downloads any dependencies for your module.

Extending Perl: the CPAN

The following two modules (part of the standard Perl distribution) are worth using:

- `autodie`
- `Data::Dumper`

```
1 use autodie;
2 open(FH,'<',$file); #Automatically safe
```

```
1 use Data::Dumper;
2 my %hash = (key1 => 'val1', key2 => 'val2');
3 print Dumper(%hash); #Easier than setting up the print yourself
```

The Docs

- Complete Perl documentation is hosted at perldoc.perl.org
- This site contains lots of helpful tutorials besides the formal documentation.
- If installed on your system, you can use the `perldoc` command to read the same documentation offline, man-page style.
- For example, the regex tutorial:

```
[user@host]$ perldoc perlretut
```

- To get help on a certain function:

```
[user@host]$ perldoc -f glob
```

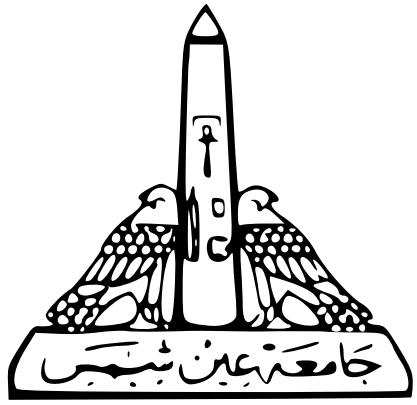
Homework

Homework

- Use any Verilog/VHDL netlist from your own work (or download one). Add VSS and VDD ports to all modules defined in that netlist.
- Write a script that would prompt the user for process corner names (e.g. tt_mos, ss_mos, ff_cap,...) and generate a CSV list of all possible corner combinations and print it to a file. This can be a basis for PVT run script.
- In the previous you homework, I asked you to think of ways Perl can help with your work. Start implementing one of them.

End of Session II.

Now you can start using Perl!



A Perl Crash Course

Session III

Joseph Samy

Ain Shams University

Feb. 6, 2013

Session III

Beyond the Basics

Session III: Beyond the Basics

Save Yourself Some Typing

- Some people like to do the minimum possible typing in Perl.
- You'll get used to the resulting code and maybe prefer it.
- Here are some rules that can help you save typing:
  - **No semicolon** is needed after the last statement in a **block**.
  - Function call **parentheses are optional** unless they would change behavior. Example:

```
1 print "Hello\n";
2 open FILE, '<', $infile or die "Fatal error!\n"
```

  - **Use** (but don't abuse) **$_ and @_**.
  - The value of the last evaluated expression in a subroutine is **automatically the return value**.
  - **You don't actually need the foreach loop**: the for loop acts in the same way:

```
1 foreach my $x (0..9){} #parentheses necessary
2 for my $x (0..9){} #equivalent to above
```

Save Yourself Some Typing

- Recall that we can use the more English-like form of conditionals:

```
1 print $x if $x > 5; #parentheses optional. Why?
2 print "Hello" unless $done;
```

- Loops can also defined this way.
- In this case, you can't define a custom iterator for the loop: you have to rely on $_as the implicit iterator:

```
1 print foreach @arr1;
2 #OR:
3 print for @arr1;
4
5 $i++ while $i<100;
```

More Useful Functions

In this section, we cover some more useful (and advanced) Perl functions:

- `map`
- `grep`
- `eval`
- `uc`, `lc`, `ucfirst` and `lcfirst`

More Useful Functions

map

- This function is used to transform a given list to a new list (of possibly different size) according to some rule(s).
- The syntax is @b = map { } @a
- What happens is:
  1. Perl iterates through @a, **aliasing** each element to $_
  2. The statements in the { } block are executed.
  3. The value of the last statement is pushed to the array on the right.

```
1 @squares = map { $_ * $_ } @numbers;
2 @squares = map { $_ > 5 ? ($_ * $_) : () } @numbers;
3 @b = map {s/one/two/g} @a #Wrong!
4 @b = map {s/one/two/g; $_} @a #Right (but be careful!)
```

More Useful Functions

grep

- Don't confuse with the Unix `grep` utility.
- The syntax is @b = grep { } @a
- The operations performed are nearly the same as those of map except that @b gets those elements of @a for which the last statement inside the { } block evaluates to true.

```
1 @b = grep {$_ > 5} @a;
2 @sq5 = map {$_ ** 2} grep {$_ > 5} @nums;
```

- Read such statements from right to left!

More Useful Functions

eval

- This function is used for two purposes:
  1. Trapping exceptions (like dividing by zero)
  2. Evaluating code from a string!

```
1 my $x = eval {5 / $y};
2 die $@ if $@ #$@ is where eval places any errors
3 eval "print \"Hello\" ";
```

- The second form can be very dangerous. Imagine the following catastrophe:

doom.pl

```
1 #!/usr/bin/perl
2 use strict; use warnings;
3 eval $ARGV[0];
```

```
[root@host /] # ./doom.pl "system(\" rm -rf * \")"
```

More Useful Functions

uc, lc, ucfirst and lcfirst

- These are very useful functions for changing the capitalization of strings.
- uc turns all the characters in its argument to uppercase and ucfirst turns only the first character of its argument to uppercase (for capitalizing names).
- lc and lcfirst work similarly for lowercase.

```
1 my @text_tosearch = map {lc} @raw_text;
2 push @b,ucfirst for qw(john smith);
```

More Useful Functions

Stop and Think

- What would map {s/one/two/} @arr do?
- How can we fix the mistake in @b = map {s/one/two;$_} @a?

Here Documents

- Sometimes you may need to prepare a very large string (e.g. to print it to a file afterwards).
- It's very inconvenient to define large strings in the conventional way.
- Use "here document" syntax instead:

```
1  my $heredoc = <<ENDSTRING;
2  This is a very long string.
3  You can interpolate variables here.
4  ENDSTRING
5  print OUTFILE $heredoc;
6  #OR
7  print OUTFILE <<'EOF';
8  This is another long string, without variable interpolation.
9  This is an amount in US dollars: $52
10 EOF
```

Sorting

- Perl uses a sort function to enable very flexible sorting of lists.
- When used simply as in

```
1 my @sorted = sort qw(orange apple banana pear);
```

  it sorts the list in normal lexicographical (alphabetical) order so that @sorted now contains qw(apple banana orange pear)
- This means that sort (1 .. 30) will not behave as you might think.
- Use the more general form for more advanced sorting:

```
1 sort { } @list; #Inline sub
2 #OR:
3 sort &subroutine_name @list; #Named sub
```

Sorting
The Sorting Subroutine

- The subroutine used for sorting needs to tell Perl how to compare two items of the list using the two special variables $a and $b.
- The subroutine must then return one of three values to show how the two variables are compared:
  1. Any value less than 0 if $a comes before $b
  2. Any value greater than 0 if $a comes after $b.
  3. 0 if $a and $b are equivalent.
- Perl has two special operators for numerical and string sorting:
  1. The so-called spaceship operator: <=> for numerical comparisons
  2. The cmp operator for string comparisons.
- For example, the following statements are equivalent:

```
1 @sorted = sort @list;
2 @sorted = sort {$a cmp $b} @list;
```

Sorting
Non-Standard Sorting

- Simple numerical sorting:

```
1 @num_sorted = sort {$a <=> $b} @list
```

- Descending numerical order:

```
1 @desc_order = reverse sort {$a <=> $b} @list
2 #Or:
3 @desc_order = sort { $b <=> $a } @list
4 # $a comes before $b when $b < $a
```

- Sorting array indices:

```
1 @ind_sorted = sort {$list[$a] cmp $list[$b]} (0 .. $#list);
```

Sorting
Non-Standard Sorting

- What does this do:

```
1  #!/usr/bin/perl
2
3  use strict;
4  use warnings;
5  use Data::Dumper;
6
7  @sorted_keys = sort &sort_people (keys %people);
8  print Dumper($people{$_}) for @sorted_keys;
9  sub sort_people
10 {
11   my $val = $people{$a}{name} cmp $people{$b}{name}
12   $val ?
13         $val : $people{$a}{age} <=> $people{$b}{age}
14 }
```

Defined-OR

- Suppose you want to open a file in your script.
- You decide to either take the filename from the command line or, if no filename is given, use a default one.
- How do you tell that to Perl?

```
1 my $filename = $ARGV[0] || 'default_name.txt';
2 #Or:
3 my $filename = $ARGV[0];
4 $filename ||= 'default_name.txt';
```

- Looks good, but isn't robust enough. What if 0 is a valid value for the command line argument?
- What we want to do is check **definededness** rather than **truth**:

```
1 my $filename = defined $ARGV[0] ?
2                          $ARGV[0] : 'default_name.txt';
```

Defined-OR

- Ever since Perl 5.10, there's a new operator called the defined-OR operator to carry out this action more easily:

```
1 my $filename = $ARGV[0];
2 $filename //= 'default_name.txt';
3 #Cascading (also possible with ||):
4 $filename = $ARGV[0] // $a[2] // $b // 'default_name.txt';
```

Smart Matching

- The smart match operator (v5.10.1 and above) is a special operator whose behavior is different depending on its operands.
- The smart match operator is two successive tildes: ~~
- One of the most common uses for smart matching is checking whether a scalar exists within an array:

```
1 print "Exists\n" if ($a ~~ @b);
```

- Smart matching a string and a regex is like regular regex matching.
- Smart matching an array and a regex returns true if any element in the array matches the regex.
- Full documentation of smart matching behavior can be found at

```
[user@host]$ perldoc perlop
```

Smart Matching

given...when

- Smart matching is used by a special construct called given .. when which is similar in behavior to the switch .. case statement of C/C++.
- Instead of simple comparisons, it uses smart matching:

```
1 given($string)
2 {
3   when (/^[0-9]+$/) {print "A number!\n"}
4   when (/^\w+$/) {print "A word!\n"}
5   when (/^\s+$/) {print "Blank!\n"}
6   default {print "Garbage!\n"}
7 }
```

References

- References are somewhat like the pointers of C/C++.
- They can be used to pass arguments to subroutines when you want your data to be modifiable from within a subroutine. This is called a **pass by reference**.
- References can be used to build arbitrarily complex data structures (array of arrays, hash of hashes, array of references to hashes of hashes,...)
- You take a reference to a variable by preceding its name with a backslash.
- A reference to any type of variable is stored in a scalar variable.

```
1 my $scalar_ref = \$sc;
2 my $array_ref = \@arr;
3 my $hash_ref = \%hash;
```

References

Dereferencing

- Dereferencing refers to the process of accessing the contents of a reference.
- For example, dereferencing an array ref, means accessing the array elements using the reference.
- Dereferencing can be done by enclosing the reference in braces preceded by the appropriate variable type prefix:

```
1 my $sc = ${$scalar_ref}
2 my @arr = @{$array_ref}
3 my %hash = %{$hash_ref}
```

- If the thing being dereferenced is a simple scalar variable, the curly braces can be dropped:

```
1 my $sc = $$scalar_ref
2 my @arr = @$array_ref
3 my %hash = %$hash_ref
```

References
Dereferencing

- The prefix to the curly braces may sometimes change depending on what you're trying to do.
- The rule for prefixing the curly braces is that you should use the prefix that you would use if the curly braces were replaced by a variable name.

```
1 my $element = ${$array_ref}[0];
2 # Like my $element = $arr_name[0];
3 my $entry = ${$hash_ref}{key};
4 # Like my $entry = $hash_name{key};
```

- You can also use arrows for dereferencing:

```
1 my $element = $arr_ref->[3];
```

References

Anonymous Variables

- Sometimes you may need intermediate variables for processing but don't want to bother naming them. That's what anonymous variables are for.
- You can have anonymous arrays, hashes or subroutines.
- Anonymous variables are stored as references.
- Anonymous arrays are created by enclosing data in [ ], anonymous hashes by enclosing data in { } and anonymous subs by enclosing statements in sub { }.

```
1 my @arr_of_arr= ([1 2 3] , ['a'..'z']);
2 $arr_of_arr[1]->[1]; #=b
3 #Or:
4 $arr_of_arr[1][1]; #arrows can be dropped between subscripts
```

References

Anonymous Variables

```
1  my @arr_of_hash = ( #You can make a hash of hashes, too
2  {
3     name => 'john',
4     age => 23,
5  }
6  ,
7  {
8     name => 'smith',
9     age => 25,
10 } );
11 my $name = $arr_of_hash[0]{name};
12 #Or:
13 $name = $arr_of_hash[0]->{name};
14 #Or:
15 $name = ${$arr_of_hash[0]}{name}
```

References

Anonymous Variables

- Subroutines can be variables, too!
- Example: a "dispatch table":

```
1 my %actions = ( abort => sub {exit} ,
2                 speak => sub { print "Hi this is $0\n" });
3 my $input = $ARGV[0];
4 exists $actions{$input} ?
5         $actions{$input}() : die "Error: unkown action\n";
```

- Arrays of references are way more flexible than you think:

```
1 my @a = ( ['a' .. 'x'], sub{print "Hi\n"}, { name => 'john',
      age => 20} );
2 print $a[0][-1],"\n"; #prints x
3 $a[1](); #prints Hi
4 print $a[2]{age}; #prints 20
```

References

You Try

- Write a script that prompts the user for personal information then stores it as a hash of hashes keyed by record number.
- Print the data you read into a CSV formatted file. Do **not** use Data::Dumper

Regular Expressions Revisited

Lookaround

- We mentioned before how regex anchors (^,$ and \b)are **zero-width assertions**: they express a *position* in the string rather than actual characters.
- Lookaround constructs are another kind of zero-width assertions.
- Use lookaround to look for a position near a certain pattern of characters.
- There are four types of lookaround:
  1. **Positive lookahead** (find a position followed by a pattern)
  2. **Negative lookahead** (find a position not followed by a pattern)
  3. **Positive look behind** (find a position preceded by a pattern)
  4. **Negative look behind** (find a position not preceded by a pattern)

Regular Expressions Revisited

Lookaround

- The following table summarizes lookaround syntax:

Lookaround	Syntax	Notes
+ve lookahead	(?=<pattern>)	Parentheses are non-capturing
-ve lookahead	(?!<pattern>)	
+ve look behind	(?<=<pattern>)	Variable length look behind isn't supported
-ve look behind	(?<!<pattern>)	

- Example:

```
1 my $x = 'catastrophe';
2 $x =~ /(cat(?=\w{3}))/; #Positive lookahead
3 print $1; #prints cat NOT catast!!
```

Regular Expressions Revisited

Non-Capturing Parentheses

- Sometimes you need to use parentheses in your regex just for grouping. In other words, you don't want them storing the match.
- Having your parentheses not capturing their contents (when you don't need them) is also more efficient.
- To do this, you simply add ?: at the start of your parentheses:

```
1 my $number = 182.24;
2 $number=~ m/([1-9][0-9]*)(?:\.([0-9]+))?/;
3 my ($integer,$decimal) = ($1,$2);
```

Regular Expressions Revisited

Pre-Compiled Regex

- When there's a part of the regex you would be using a lot, it's best to compile it only once.
- Also helps readability (modular regex).
- You specify this to Perl by using the qr// operator.
- Example on constructing a scientific notation number:

```
1 my $float = qr/[1-9][0-9]*(?:\.[0-9]+)?/;
2 my $power = qr/[0-9]+/; #Can have leading zeros
3 my $sci_num = qr/[-+]?$float[eE][-+]?$power/;
```

Regular Expressions Revisited

More Regex Modifiers

- As regexes grow they become increasingly difficult to follow. The /x modifier allows you to enter comments (preceded by # as usual) within the regex and add arbitray white space to it:

```
1 my $float = qr/^ #Start of string
2                  ([1-9][0-9]*) #Integer part (in $1)
3                  (?:\.[0-9]+)? #Optional fraction
4                  $ #End of string
5                /x;
```

- What if you want your regex to match actual whitespace? Simply escape it:

```
1 my $regex = qr/Hello,\ world! #Literal space
2              /x;
```

Regular Expressions Revisited

Interpolating Variables

- You know that we can interpolate variables into regexes.
- What if you want the variable to be interpolated literally (i.e., where $*$, $+$,... have no special meaning)?
- Perl defines two special escape sequences for this: \Q and \E. Anything between these two escapes pays no attention to regex metacharacters.
- If no \E is supplied, the rest of the regex is interpreted literally.

```
1 my $money_amount = '$52.23';
2 $string =~ /(?:Sh|H)e paid \Q$money_amount\E exactly$/;
```

Regular Expressions Revisited

Interpolating Variables

- When performing a regex substitution, you might want the substituted part to be computed during runtime.
- This can be done by using the /e modifier (short for eval).

```
1 my @a = (0..9);
2 s/([0-9])/$1+1/e for @a;
```

- This modifier can be repeated several times. For example, /ee means to evaluate the substituted expression twice before substituting it:

```
1 %subs = ( 1 => "print 25" );
2 $i = 0;
3 $string =~ s/youraction/$subs{++$i}/ee;
```

Regular Expressions Revisited

(Ir)regular Expression Insanity

- Three special regular expression constructs:
  1. (?{..code..}): Execute code within a regex.
  2. (??{..code..}): Execute code within a regex and use the result as part of the regex to match.
  3. (?(condition)then | else) Checks a condition and matches against the then regex or the else regex according to the condition. Usually the condition is a single number. (?(1)cat|dog) means if capture group 1 (i.e. the parentheses tied to $1) has matched, then match cat; otherwise, match dog.
- I invite you to read more about them in

```
[user@host]$ perldoc perlretut
```

```
[user@host]$ perldoc perlre
```

Regular Expressions Revisited

You Try

- Use split to split a slurped file into lines, preserving the newline at the end of each.
- What would the regex m/^Trick Question$/x match?

File Tests

- File tests are tests used to determine the properties of files.
- Used to ensure a file has the right properties before working with it.
- The most important tests:

Test	Meaning
-f	Is a file
-d	Is a directory
-r	Is readable
-w	Is writable
-x	Is executable

- All the tests return a boolean value and all have $_ as default operand.

File Tests

- Examples:

```
1 -r $file or die "File isn't readable\n";
2 -f and print "$_ is a file\n" for(@files);
3 my @directories = grep{-d} glob '*';
```

Going Beyond These Sessions

- Recommended O'reilly Books:
  1. Learning Perl
  2. Intermediate Perl
  3. Mastering Perl
  4. Programming Perl (the Camel book)
  5. Mastering Regular Expressions
- Recommended websites:
  1. perldoc.org
  2. perlmonks.org

Homework

Homework

- Explore the LWP, HTTP::Request and WWW::Mechanize modules.
- Almost everything can be automated. Look for ways to automate all your tasks (e.g. using RTF for Word)
- Explore the Tk module for making GUIs.
- Implement some of the tasks you thought of for the first homework.

End of Session III.
Now you can begin mastering Perl on your own!