CS2043 - Unix Tools & Scripting
Additional Lecture
Intro to Perl
Spring 2015

Instructor: Nicolas Savva

March 4th, 2015

1 based on slides by Hussam Abu-Libdeh, Bruno Abrahao and David Slater over the years
Perl is an acronym, short for Practical Extraction and Report Language. It was designed by Larry Wall as a tool for writing programs in the UNIX environment.

Why Perl?

- Perl has the power and flexibility of high level languages like C.
- Like shell scripting, Perl does not require a special compiler and linker to use. This makes Perl ideal for producing quick solutions to small programming problems.
- Perl provides all the features of sed and awk plus features not found in either.

In short, Perl is as powerful as C but as convenient as awk, sed and shell scripting.
By convention, we will use the .pl extension for our perl scripts (foo.pl)

To invoke the Perl interpreter we can type perl foo.pl

Or we can start our script with the special character `#!` followed by where to find perl

Perl is usually at /bin/perl (cygwin), /usr/bin/perl (ubuntu and csug machines) or /usr/bin/local/perl.

We can also run perl commands directly by typing perl -e 'cmds'.
```perl
#! /usr/bin/perl
#
#$inputline = <STDIN>;
print $inputline;

$ ./babyperl.pl
Here is my line of text
Here is my line of text

#$hello.pl
print "Hello World!\n";

$ ./hello.pl
Hello World!
$```

Baby Tiny Simple Silly Perl Scripts
A Perl script is a sequence of statements (each statement is one task for the Perl interpreter to perform).

Perl programming is **not** shell programming. You cannot use shell commands in Perl scripts (directly).

Comments go anywhere and begin with a `#`

Perl does not care about whitespace and new lines, all of these are equivalent

```perl
$input=5;
$input = 5;
$input  = 5 ;
$input =
  5 ;
```
Perl recognizes three types of values: integers, strings and objects (we won’t deal with objects)
We can write strings using either single or double quotes
A string in single quotes is taken to mean exactly what it is
a string in double-quotes is first interpreted (similar to with bash scripting)
scalar variables in Perl can be integers and strings and are required to start with the $ character (different from shell scripting!).

```
$i = 10;
$world = 'hello';
print $i;
10
```
All variables must begin with $ followed by a letter.

All Perl statements end with a ;

Double quoted strings perform variable substitution

```
$input=5;
print 'the value of $i is', " $i.";
```

Prints:

the value of $i is 10.
print

print is a built in command with the syntax:

print expr,expr,expr...,expr;

where expr can be an int, a string or in general any expression yielding such. print does not put newlines after printing you must explicitly add them using "\n"

print "This is the value ";
print 'of integer variable $i: ', $i, "\n","Exciting!";

This is the value of integer variable $i: 10

Exciting!
Perl Baby Steps...

- \n must be in double quotes, if you don’t quote it the following occurs:

```perl
print \n
SCALAR(0x10423ad8)
```

- We can perform scalar operations as expected:

```perl
$a = "10" ; $b = $a + 2 ; $c = $b + " 34 " ;
```

Assigns $a 10, $b 12 and $c 46.

- Perl automatically converts between strings and ints as needed. If no meaningful conversion to an int can be done then 0 is used.
- Perl Arithmetic: +, -, *, /, %, ** (exponent)
- Perl Assignment operators: =, +=, -=, *=, /=, %=, **=
- Perl does not understand complex numbers
Some Perl operations require strings

Basic Example: $foo . $bar concatenates to $foobart.

$a = 5;
$b = $a + 10;
$c = $b . "200";
$d = $c + 10;

Assigns $a 5, $b 15, $c 15200 and $d 15210.
The following are integer comparison operations:

- <
- >
- <=
- >=
- ==
- !=
- <=> - 1 if first value greater, -1 if second is greater, 0 if equal.

I bet you can all guess what the rest do!
The following are string comparison operations:

- **gt** (greater than)
- **ge** (greater than or equal)
- **lt** (less than)
- **le** (less than or equal)
- **eq** (equal)
- **ne** (not equal)
- **cmp** (compare returning 1, 0 or -1)

Exciting isn’t it?
Strings are not ints!

String equality is not the same as integer equality:

- "15" == ' 15 ' - true
- "15" eq "15 " - false
- $a \text{ || } b$ - logical or: true if either is nonzero
- $a \text{ && } b$ - logical and: true only if both are nonzero
- $! a$ - logical not: true if $a$ is zero
- $a \text{ or } b$ - same as $||$
- $a \text{ and } b$ - same as $&&$
- not $a$ - same as $!$
- $a \text{ xor } b$ - logical xor: true if $a$ or $b$ but not both

or, and and not have a lower precedence.
those pesky conditional statements

if

if (comparison expression) {
    statement;
    ...
    statement;
}
elif (comparison expression) {
    statement;
    ...
    statement;
} else {
    statement;
    ...
    statement;
}

The {} and ( ) are mandatory!
Remember Perl does not care about newlines and spacing.
while

while (comparison expression) {
    statement;
    ...
    statement;
}


#!/usr/bin/perl
$done = 0;
$count = 1;
print "This line is printed before the loop starts.\n";
while ($done == 0) {
    print "The value of count is", $count, "\n";
    if ($count == 3) {
        $done = 1;
    }
    $count = $count + 1;
} print "End of loop.\n";

Prints:

This line is printed before the loop starts.
The value of count is 1
the value of count is 2
The value of count is 3
End of loop.
until...

until (comparison statement) {
    statement
    ...
    statement
}

Exciting.
Like in bash scripting there are multiple ways to do for loops:

```bash
for variable (value,...,value) {
    statement;
    ...
    statement;
}

for (setup; condition; increment) {
    statement;
    ...
    statement;
}
```
for $i (2,3,5,7,11,13,17,19) \{
    \text{\quad print } \text{"Here's a prime: $i\n"};
\}

for ($i = 0; i \leq 30 ; i++) \{ 
    \text{\quad print } \text{"$i\n"};
\}
In perl we can do both pre and post incrementing.

```perl
$var1 = 43;
$var2 = ++$var1;

Then $var1 is 44 and $var2 is 44,

$var1 = 43;
$var2 = $var1++;

Then $var1 is 44 but $var2 is 43!
```
The Conditional Operator:

condition to test ? value if true : value if false

The Comma Operator

Use the comma operator to guarantee what is before the comma is evaluated first

$var1 += 1 , $var2 = $var1;
String Operators

Autoincrement with Strings:

$stringvar = "abc";
$stringvar++;
Then $stringvar is abd.

$string-- does not work with strings

The repeat operator:

$string = "t" x 5;
sets $string to tttttt

Concatenation and Assignment

$a = "be";
$a .= "witched";

$a is now bewitched.
Regular Expressions exist in Perl and have their own syntax. First example is with testing:

```
string =~ /regexp/
```

- `c` - matches character `c`
- `.` - matches any one character, except newline
- `?` - matches zero or one occurrences of the previous character
- `*` - matches zero or more occurrences of the previous character
- `+` - matches one or more occurrences of the previous character
- `[` - matches anything inside the `[` (can do ranges)
- `[^... ]` - matches anything not inside the `[`
\d - matches a digit
\D - matches nondigit
\w - matches a "word" (letter, digit or an underscore)
\W - matches non-word
\s - matches whitespace character
\S - matches non-whitespace character
More Reg Exp Rules

- ^ - matches beginning of string
- $ - matches end of string
- \b - word boundary
- \B - not on word boundary

/\bdef/ - matches def at the beginning of a word
/def\b/ - matches def at the end of a word
/\Bdef/ - matches def as long as it is not the beginning of a word

- ( ) - store in memory as $1, $2 etc
- $& - holds the matched pattern

There are more but thats good for now.
Pattern Matching Options

- g - match all possible matches
- i - ignore case
- m - treat string as multiple lines (in this case ^ and $ also match beginning/end of lines)
- s - treat string as single line (in this case you can match the newline character)
- x - ignore white space in pattern (easier to read then)

Options go after the second /

@matches = "balata" =~ /a/g;

Then the array @matches is ("ba", "la", "ta").

/^[A-Z][a-z]+ \ [A-Z] [a-z]+ /x

Matches names such as David Slater.
We can also use Reg Exps to do Replacement:

```
s/pattern/replacement/
```

```perl
$string = "abc123def123";
$string =~ s/123/456/;

Then $string is abc456def123 (since we did not use the flag g.

```perl
$string = "Slater, David";
$string =~ s/(.+), (.+)/$2 $1/

Then $string is David Slater.
Perl has a translation operator just like `tr` in bash, in fact it is called the same thing!

```
tr/string1/string2
```

```perl
$string = "abcdefgghicba";
$string =~ tr/abc/def/;
```

Then $string is defdefghifed. Note that it is a character by character translation!

```
tr/A-Z/a-z/     translate upper to lower case characters.
```

Options for `tr`:
- `c` - translate all characters not specified
- `d` - deletes all specified characters
- `s` - replace multiple identical output characters with a single characters.
Perl 5 can do even more:

- 
  (?<c>pattern) - <c> represents a single character representing the extended pattern-matching capability being used and pattern is the pattern/sub pattern
effected
- 
  /(?;string)/ - don't store string in memory.
- 
  /(?;a|b|c)(d|e)f\1/ - matches one of a,b or c, one of d or e, f, whichever of
d or e was matched earlier.
- 
  /(?option)pattern/ - options include i (ignore case), m (treat pattern as
  multiple lines) and x (ignore white space).

/a-z\+/i is equivalent to /(?i)[a-z]+/
Extended Pattern Matching

- `/abc(?=def)/` - matches abc only if it is followed by def (Positive Look Ahead)
- `/abc(?!def)/` - matches abc only if it is not followed by def (Negative Look Ahead)
- `/(?i)[a-z]{2,3}(?#` matches two or three alphabetic characters)```/` - comments
Perl can do standard arrays and associative arrays (dictionaries). We’ll talk about associative arrays later.

An array variable is just like a scalar variable except it starts with @:

```perl
@names = ("David", "James", "Joan", "Sarah");
print "The elements of names are @names\n";
print "The first element is $names[0] \n";
print "There are ", scalar(@names)," elements in the array\n";
```

Prints:

The elements of @names are David James Joan Sarah
The first element is David
There are 4 elements in the array
Arrays

- When referring to more than one variable use @array
- When referring to a single element use $array[i]
- $array would refer to a scalar variable called array.
- @array[i,j,k] to refer to multiple elements
- @array[i..j] to refer to a range of elements
- $#array to refer to the last index
- $scalar = @array will set $scalar to the length of @array.
Arrays and for loops

Looping over elements in an array is easy

```plaintext
for $x (@array) {
    print "$x \n;
}

foreach $x (@array) {
    print "$x \n;
}

Both are equivalent.
```
The infamous $_

We can do this even shorter by using $_, which is the default input and pattern search variable.

```perl
foreach (@array) {
    print $_;  
}
```

Or

```perl
foreach (@array) {
    print;
}
```

$_ is one of the most used special variables. It is implicitly assigned to input streams, subroutine variables, pattern-searching space and the default iterator variable in loops. We’ll look at examples on a case by case basis. Many functions default to $ when no argument is given.
We can end loops early by using `last`

```perl
@names = ("Mr Slater", "Ms Jones", "Dr Heinlein", "Sir James");

foreach $person (@names) {
    print "$person\n";
    last if $person = /Dr/;
}
```

**Prints**

Mr Slater
Ms Jones
Dr Heinlein
Changing Array Elements

You can edit an array element just as in bash

```bash
@myarray=(1,2,3);
$myarray[2]=5;
print "@myarray";
```

Prints

```
1 2 5
```

We can also add elements to an array in the same manner, but it does this sequentially:

```bash
@myarray=(1,2,3);
$myarray[10]=5;
print "@myarray\n", "$#myarray";
```

Prints

```
1 2 3 5
10
```
Changing Array Elements

- **push** adds elements to the end of an array

  ```javascript
  @names = ("David", "James", "Joan", "Sarah");
  @morenames = ("Mark", "Jason", "Jamie");

  push (@names, "Something", @morenames );
  ```

- **pop** removes the last element and returns it
- **shift** removes the first element and returns it
- **unshift** adds value to the beginning of array
Array functions

- sort(@array) sorts the array alphabetically
- reverse(@array) reverse the order of an array
- These do not change the original array, so you need to do
  ```perl
  @array = sort(@array);
  ```
- chop(@array) removes the last character from each array element
  ```perl
  @array = ("rabbit", "12345","quartz");
  chop(@array);
  print "@array";
  ```
- Prints rabb 1234 quar
$string = \text{join}(\text{" "}, @\text{myarray});$ converts an array to a string with spaces between elements

@array = \text{split}(/:::/, \text{string}); - splits a string an array by using :: as a delimiter. Can you regexps!

$string = \text{"Hi::to::you::class"};$
@array = \text{split}(/:::/, $string);

Assigns (Hi, to, you, class) to @array.
Splicers everywhere!

We can work with the middle of an array using splice

- `@middle = splice(@array, i, j)` - assigns j elements of @array starting with element i to @middle and removes them from @array.
- `splice(@array, i, 0, @secrarray)` - adds the elements @secrarray to @array in place i.
- `splice(@array, i, j, @secrarray)` - replaces elements of @array with elements of @secrarray.
- `splice(@array, i, 1)` - removes element at position i.
To open a file, simply type

```perl
open (filevar, filename);
```

`filevar` is called the file handle or file variable and must start with a letter and contain only letters, digits and underscores.

It is a good idea to close a file when you are done using it, to do so type

```perl
close (filevar);
```

Perl automatically closes files when the program terminates or when you open another file with the same `filevar`. 
Options for opening files

Open defaults to read mode, where you can only read the context of a file.

○ `open (outfile, "~/somedirectory/outfile");` - open in write mode
○ `open (outfile, "~/somedirectory/outfile");` - open in append mode
○ When you open a file for writing, it destroys any existing content
○ You cannot read from and write to the same file at the same time
○ When you open in append mode, you cannot read from the file
○ `open(filevar, <filename>);` - also opens a file in read mode
open returns 1 if it is successful and zero if it is not, so it is easy to only run code if open is successful:

```c
if ( open (MYFILE,"foo.txt")) {
    commands to run if opening succeeded;
}
```
Once we have opening a file, we would like to be able to read it wouldn’t we? To do so we use the angle bracket operator:

$nline = <MYFILE>;$

This reads in the first line of MYFILE and saves it to $nline.

The angle brackets <> read data from a filehandle. How much it reads depends on context. Filehandles are not limited to files, they are also how we interact with things like the console and sockets.
#! usr/bin/perl
#
# This script reads in the entire contents of a file
# line by line and prints each line.
#
if (open(MYFILE, "somefile")) {
    $line = <MYFILE>;
    while ($line ne "") {
        print $line;
        $line = <MYFILE>;
    }
}

- Every time you run $line = <MYFILE> it reads the next line.
We can read in the entire file even shorter by using 

```perl
while (<MYFILE>) {
    $line =$_;
    print $line;
}
```

$_ is the Default Input and Pattern Searching Variable
Yes! You can easily read it into an array:

@myarray = <MYFILE>;

And with a little more work we can read into a scalar variable:

{  
local ($/)
open(MYFILE,myfile);
$text = <MYFILE>;
}

Here $/ is input record separator and we have locally made it undefined.
We can change the input record separator if we want for some reason. For example if you have a file containing something like

Tea::Beer::Wine::Pizza::Catfood::Dogfood

We can read it in one item at a time and do something to each element.

```perl
#!/usr/bin/perl

$SHOP="shop.txt" ;
$=/"::";

open(STUFF,$SHOP);
while (<STUFF>) {
    print "$_\n" if $_ =~ /[Bb]eer/;
}

Prints

Beer::
```
On the last slide we saw that when we change the newline character, it is included in the variable when we print. We can avoid printing it by using `chomp:

```
#!/usr/bin/perl

$SHOP = "shop.txt";
$/ = ":";

open(STUFF, $SHOP);
while (<STUFF>) {
    print "$_\n" if $_ =~ /[Bb]eer/;
}
```

**Note:** If we are writing a larger script and only want to change it in one place it would be good to locally change `$/` as we saw earlier. We will see more useful applications of `chomp` in, oh lets say 2 slides?
We can also quit our program if reading in fails by using `die`:

```perl
open(MYFILE, file) or die "sudden flaming death\n";
```
Perl has built-in handles corresponding to standard input and standard output. Thus if we do

```perl
$myvar = <STDIN>;
```

We will read in one line of input. The only problem is it also reads in the newline character when you push enter! To solve this we usually just use `chomp`, which removes the last character only if it is a newline (actually it removes the last character if it matches `$/`).

```perl
$myvar=<STDIN>;
chomp($myvar);
```

or

```perl
chomp($myvar=<STDIN>);
```
We can also read in any bash command by using a pipe:

```
open(MYVAR, "cat somefile | sort |" );
```
To write to a file we use the print command

```perl
open(OUTFILE,">outfile");
print OUTFILE "Here is an outline line.\n";
close(OUTFILE);
```

```perl
#!/user/bin/perl
# This file opens file1 and writes it to file2.
open(INFILE,"file1") or die ("sudden flaming when opening file1 \n");
open(OUTFILE,"file2") or die ("sudden death when opening file2 \n");
while(<INFIL>) {
    print OUTFILE \$_;
}
```

Note: We could have skipped \$_.
print operators on a lists, so if you do

print <INFILE>

Perl will print the entire file. Thus the last script could be simplified to

```perl
#!/user/bin/perl
# This file opens file1 and writes it to file2.
open(INFILE,"file1") or die ("sudden flaming when opening file1 \n");
open(OUTFILE,"file2") or die ("sudden death when opening file2 \n");
print OUTFILE <INFILE>;
}

This is classic in Perl, many ways to do the same thing.
print "Here is output \n";

and

print STDOUT "Here is output.\n";

Are equivalent
STDOUT is a filehandle, it is perfectly valid to re-open any existing filehandle. When you do so, the proceeding one is first closed and a new one is opened. Let's look at an example
So now lets do something useful. Lets say we have a file with some text in it and we want to do some operation. For the sake of being superfluous, lets say we want to change "away" to "yellow", but only if it occurs twice in a line. Here we go...

```perl
#!/usr/bin/perl
#nonabusive version
$file = "test.txt";

#open the file or die with grace
open (FILE,"<$file") or die "Can’t open $file: $!
";
@lines = <FILE>;
close FILE;

#Open same file for writing
open (OUTFILE, ">$file") or die "Can’t open $file: $!
";

# loop over the lines and do the job
for $line (@lines)
{
    $line =~ s/(.*away.*)away/$1yellow/;
    print OUTFILE $line;
}
```
#! /usr/bin/perl
# Now we abuse Perl

$file = "test.txt";

#open the file or die with grace
open (FILE,"<$file") or die "Can’t open $file: $!
"
@lines = <FILE>;
close FILE;

#Open same file for writing;
open (STDOUT, ">$file") or die "Can’t open $file: $!
";

# loop over the lines and do the job
for (@lines) {
  s/(.*away.*)away/\$1yellow/;
  print;
}

Thankfully file testing for Perl is similar to bash

- `-d` : checks if directory
- `-e` : checks if exists
- `-l` : checks if it is a symbolic link
- `-o` : checks if file is owned by the user
- `-r` : checks if reading
- `-w` : checks if writable
- `-z` : checks if file is empty

And there are many more.
#!/usr/bin/perl

# This script opens file1 and writes it to file2
# if it does not already exist.

open(INFILE,"file1") or die "sudden flaming when opening file1 \n";
!(-e "file2") or die "file2 exists!\n";
open(OUTFILE,">file2") or die "sudden death when opening file2 \n";
while(<INFILE>) {
    print OUTFILE;
}
#!/usr/bin/perl

$file=ARGV[0];
my (@description, $size);
if (-e $file)
{
    push @description, 'binary' if (-B _);
    push @description, 'a socket' if (-S _);
    push @description, 'a text file' if (-T _);
    push @description, 'a block special file' if (-b _);
    push @description, 'a character special file' if (-c _);
    push @description, 'a directory' if (-d _);
    push @description, 'executable' if (-x _);
    push @description, (($size = -s _) ? "$size bytes" : 'empty');
    print "$file is ", join(' ', @description), "\n";
}
Perl stores error codes for the last command executed in $!. So we can modify our script again

```perl
#!/usr/bin/perl
# This script opens file1 and writes it to file2
# if it does not already exist.

open(INFILE,"file1") or die "Error opening file1: $! \n";
!(-e "outfile") or die "file2 exists!\n";
open(OUTFILE,">file2") or die "Error opening file2 $! \n";
while(<INFILE>) {
    print OUTFILE;
}
```
Perl uses the array @ARGV to store commandline arguments. For example if we run

./myperlscript.pl arg1 arg2

Then we can access agr1 and agr2 as $ARGV[0] and $ARGV[1] respectively. You can also do

$numargs = @ARGV;

To get the number of arguments.
#!/usr/local/bin/perl

# Usage: perl phone.pl phonedb.txt areacode.txt

# Entries in the first argument are of the form
# Mary Smith (607) 255-0010 (607) 342-0019
# John Peter (507) 675-7241 (607) 255-5555
# The second file will contain the names of each person,
# this time last name first, and the area codes for each
# the person's phone numbers

eprint ISTS opent, "$ARGV[0]"")
     if (open(OUTFILE,">$ARGV[1]")
        while ($myline=<INFILE>)
            $myline =~s/\s+(\w+)/s+\1$/2 $1/;
            $myline =~s/\((\d{3})\)\s*\d{3}-\d{4}$/1/g);
            print OUTFILE "$myline";
        }close(INFILE);
        close(OUTFILE);
    }else {
        print "Error opening $ARGV[1]\n";
        close(INFILE);
        exit(1);
    }
}else {
    print "Error opening $ARGV[0]\n";
    exit (1);
    } X
#!/usr/local/bin/perl

Usage: perl phone.pl phonedb.txt areacode.txt

Entries in the first argument are of the form
Mary Smith (607) 255-0010 (607) 342-0019
John Peter (507) 675-7241 (607) 255-5555

The second file will contain the names of each person,
this time last name first, and the area codes for each
the person’s phone numbers

open(INFILE,"$ARGV[0]"") or die "Error Opening $ARGV[0] $!";
open(OUTFILE,"$ARGV[1]"") or die "Error Opening $ARGV[1] $!";

while (<INFILE>) {
    s/\s*\((\w+)\s+(\w+)/$2 $1/;
    s/\d{3}-\d{4}/$1/g;
    print OUTFILE;
}

close(INFILE);
close(OUTFILE);
Or we can read into an array

```perl
#!/usr/local/bin/perl

# Usage: perl phone.pl phonedb.txt areacode.txt

# Entries in the first argument are of the form
# Mary Smith (607) 255-0010 (607) 342-0019
# John Peter (507) 675-7241 (607) 255-5555
# The second file will contain the names of each person,
# this time last name first, and the area codes for each
# the person’s phone numbers

open(INFILE,"$ARGV[0]"");
open(OUTFILE,"$ARGV[1]"");

@phonearray=<INFILE>;

s/(?::\s*(\w+)\s*\(\w+\))\s*\((\d{3})\)\s*\d{3}-\d{4}/$2 $1 $3/ for (@phonearray);

print OUTFILE @phonearray;

close(INFILE);
close(OUTFILE);
```

But this requires us to store the whole array in memory.
Wait we can do this in another way?

```perl
#!/usr/local/bin/perl  
# Usage: perl phone.pl phoneldb.txt areacode.txt
# Entries in the first argument are of the form
# Mary Smith (607) 255-0010 (607) 342-0019
# John Peter (507) 675-7241 (607) 255-5555
# The second file will contain the names of each person,
# this time last name first, and the area codes for each
# the person’s phone numbers

open(OUTFILE,"$ARGV[1]") or die "Error opening $ARGV[1]
pop(@ARGV);

while (<>)
{
    s/\s*\([^\w+]+\)\s\([^\w+]+\)/$2 $1/;
    s/\([^\{3}\)]\)\s*\[^\{3\]}-\[^\{4\}]\)/$1/g;
    print OUTFILE;
}

close(INFILE);
close(OUTFILE);

<> will automatically go through everything in @ARGV. We removed the second element so it just went through the first!!
```
Angle Brackets

What the angle bracket operator is actually doing is

- When the Perl interpreter sees the <> for the first time, it opens the file whose name is stored in $ARGV[0].
- After opening the file, the Perl interpreter executes the following library function: `shift(@ARGV);
- This library function gets rid of the first element of @ARGV and moves every other element over one. This means that element x of @ARGV becomes element x-1.
- The <> operator then reads all of the lines of the file opened in step 1.
- When the <> operator exhausts an input file, the Perl interpreter goes back to step 1 and repeats the cycle again.
So if we run `perl myscript.pl stuff.txt out.txt` where `myscript.pl` contains

```
while (<>){
  print;
}
```

Will print the contents of both files, but so will

```
print while <>;
```

And

```
print <>;
```
In Perl, we can in fact open a file and write to it at the same time using $^I, the inplace-edit extension :

```perl
#!/usr/bin/perl

@ARGV="fr.txt";

$^I=".bk";

while (<>)
{
    tr/A-Z/a-z/;
    print;
}
```
#! /usr/local/bin/perl
# Usage: perl simple_dict.pl word
# Script that opens dictionary.txt which contains on each line a word and its definition, 
# and outputs all the definitions for word, and then counts the total number of such definitions.

open (IN,"dictionary.txt") or die "Cannot open dictionary.txt: $!\n";
# check word
if ($#ARGV != 0){
    print "Wrong number of arguments!\n";
}
else {
    $word=$ARGV[0];
}
# find defintions
while ($line=<IN>){
    @s=split(/ /,$line,2);
    if ($s[0] eq $word) {
        print "$s[1]"; $n+=1;
    }
}
print "\n The word $word has $n definitions \n";
seek(filevar, distance, relative_to) - filevar distance (in bytes) forward from:

- relative_to = 0 - relative to the beginning of the file
- relative_to = 1 - relative to the current position in the file
- relative_to = 2 - relative to the end of the file

Examples:

- seek(MYFILE, 0 0); - skip to the beginning
- seek(MYFILE, 80,1); - skip forward 80 bytes
- seek(MYFILE, -80,2); - skip 80 bytes from the end
tell me about it...

*tell(filevar)* - returns the distance (in bytes) between the beginning and the current position of a file.
Directories

- opendir(DIRHANDLE, EXPR) open a directory
- closedir DIRHANDLE close a directory
- readdir - reads one filename from DIRHANDLE

```perl
#! /usr/bin/perl
opendir (DIR, '.') or die "Couldn't open directory, $!");
while ($file = readdir DIR) {
  print "$file\n"
}
closedir DIR;

Print all files in the current directory
```
#!/usr/bin/perl
#
# findtex

opendir (DIR, '.') or die sudden death
foreach (sort grep(/.*\.tex$/, readdir(DIR)))
{
    print "$_\n";
}
closedir DIR;

**Note:** perl grep and bash grep are not exactly the same.
grep

@LIST = grep(EXPR, @ARRAY);
Searches each element of array for the regular expression
EXPR and returns only the elements that evaluate as true.


grepNames = grep(/A/, myNames);
Associative arrays start with the special character %. They work very similarly to associative arrays in gawk:

```bash
$names{"David" } = 1;
```

This assigns 1 to the element David in the associative array %names.

Note that we use curly brackets `{ }`.

Easiest way to add to an associative array / create one is to just assign something to it as above.
We can create an array with multiple elements by typing:

```
%ShoppingList = ("apples", 4, "bananas", "one bunch", "cereal", "cheerios");
```

Or to make it easier you can do

```
%ShoppingList = ("apples" => 4, "bananas" =>"one bunch", "cereal" => "cheerios");
```

Did I mention that Perl likes to give you multiple ways to do things?
You can copy a normal array to an associative array:

```perl
@ShoppingList = ("apples", 4, "bananas", "one bunch", "cereal", "cheerios");

%ShoppingList = @ShoppingList
```

And Perl will create the three element associative array.
Working with Associative Arrays

- $AssocArray{"some"}$ - to get the value stored in some
- `delete($AssocArray{"some"})` ; - deletes the element
- **Don’t** use `push`, `pop`, or `shift` with associative arrays because you don’t know the order elements are stored in.
The function `keys` retrieves the a list of subscripts used in an associative array:

```perl
%ShoppingList = ("apples" => 4, "bananas" => "one bunch", "cereal" => "cheerios");
@listkeys = keys(%ShoppingList);
```

Then `listkeys` contains bananas, apples and cereal.

The function `values` retrieves a list of values stored in an associative array:

```perl
@listvals = values(%ShoppingList);
```

Then `listvals` contains one bunch, 4 and

NOTE: IN both cases, the ordering is not necessarily the given one
looping with these arrays

We can easily loop using a foreach loop:

```perl
%Records = ("David" => 100, "Mark" => 55, "James" => 78);
foreach $holder (keys(%Records)) {
    Do Stuff;
}
```

This works fine, but is inefficient as it must look up the value in the array everytime you want to use it. Instead...
while (($holder, $record) = each(%Records)) {
    Do Stuff
}

Which uses the built in command each
#!/usr/bin/perl
# ls.pl
# This program runs the ls -l command on the current directory
# And prints out the files and their sizes

open(IN,"ls -l | ");
<IN>;

while ($line=<IN>) {
  @fields=split(/\s+/, $line);
}
close(IN);
print "File  \t  Size\n";
while (($file,$info)=each(%myfiles)) {
print "$file  \t  $info\n";
}
The popular command printf from C exists for Perl. So we can do things like:

printf "My name is %s and this is Lecture %d", "David Slater", 9;

Conversions:
- %% - A percent sign
- %c - A character with a number
- %s - A string
- %d - A signed integer in decimal
- %e - A floating-point number in scientific notation
- %f - A floating-point number in fixed decimal notation

You can do plenty more with printf but we won’t get into that.
You can use the `<<` to indicate the beginning of a string. So you can do things like:

```bash
$mystring= <<
This is the first line
This is the second line
This continues till the next blank line

print << END;
This is a line
This is another line

This is a line after a space
The next line ends the print statement
END
```
So far we have run bash commands by piping them into a file handle

```bash
open(CMD, cmd | cmd | cmd |);
```

This works fine, but of course we can do it other ways:

```bash
$scalarvar='ls -l';
$arrayvar='ls -l';
print system("ls -l");
print exec("ls -l");
```
That being said, Perl has been designed to do what shell scripting can do, so if you can do it with bash commands you can probably do it with perl:

```perl
opendir(DIR,".");
@files=readdir(DIR);

print "$_\n" foreach (@files);
```
In Perl, functions are called subroutines and are written

```perl
sub mysubroutine {
    some code;
    some more code;
    best code ever;
}
```

Like everything else in Perl white space does not matter.
Calling Subroutines

- When using subroutines you call it by typing `&mysubroutine`
- You can declare a scalar variable `$mysubroutine` and it will be separate
- Subroutines go **anywhere** in your script
- A common convention is to put them all at the bottom or the top.
A Subroutine will return the value of the last expression automatically, but be careful as to what the last expression is!

```perl
#!/usr/bin/perl

$total = &get_total;
print("The total is $total\n");

sub get_total {
    $value = 0;
    chomp($inputline = <STDIN>);
    @subnumbers = split(/\s+/, $inputline);
    $number = 0;
    while ($subnumbers[$number] ne "") {
        $value += $subnumbers[$number++];
    }
}

What's wrong with this?
The last expression evaluated was

$subnumbers[$number] ne ""

This value is zero (the last time it runs), and since zero is treated as the null string by print in Perl, the script prints "The total is". Fix:

```perl
#!/usr/bin/perl

$total = &get_total;
print("The total is $total\n");

sub get_total {
    $value = 0;
    chomp($inputline = <STDIN>);
    @subnumbers = split(/\s+/, $inputline);
    $number = 0;
    while ($subnumbers[$number] ne "") {
        $value += $subnumbers[$number++];
    }
    $retval=$value;
}
```
Of course, we could have just used the Perl return statement

return(some value)

instead.
Passing values to subroutines

If you call a subroutine using

&somesubroutine(value1, value2, value3),

then value1 value2 and value3 are stored in the array @.
Perl has a variety of modules that extend the core features of the Perl language.

- A module is a collection of related subroutines and data designed to be used by other programs or modules.
- Can be as simple as a collection of subroutines or have classes and objects.
- Use them with use
File::Find does a recursive directory search and performs tasks each time a file or directory matches.

use File::Find
find(\&Wanted, $dir);

find() does a recursive subdirectory search of $dir. It calls Wanted() once for each file and directory in $dir including $dir itself.

You get to write Wanted! It is an arbitrary subroutine. When it is called

- $_[ is set to the current file name
- $File::Find::Dir is set to the current directory
- $File::Find::names is set to "$File::Find::dir/$_"

Of course you can call wanted whatever you want.
#!/usr/bin/perl

use File::Find;

$dir=ARGV[0];

find(&d,$dir);

sub d {
  print "$_\n";
}

Prints all files and directories in all subdirectories
Slightly better example

```perl
#!/usr/bin/perl
# findregex.pl

use File::Find;
$expr=$ARGV[1];
$dir=$ARGV[0];

find(\&d,$dir);

sub d {
  /$expr/ or return;
  print "$File::Find::name
";
}

Usage: ./findregex.pl directory regexp
Prints all files/directories that match the regular expression
```
That's about all the Perl we have time to cover, but there is a lot more you can do with it:

- Object Oriented Programming
- Extend it with more Modules
- Work with a ton of System Functions
- And much much more