COMP519: Web Programming
Autumn 2014

Perl Tutorial: Beyond the Basics

- Keyboard Input and file handling
- Control structures
- Conditionals
- String Matching
- Substitution and Translation
- Split
- Associative arrays
- Subroutines
- References

Keyboard Input

```
#!/usr/local/bin/perl
# basic08.pl COMP519
print "Please input the circle's radius: ";
$radius = <STDIN>;
$area = 3.14159265 * $radius * $radius;
print "The area is: $area \n";
```

```
bash-2.05b$ ./ basic08.pl
Please input the circle's radius: 100
The area is: 31415.9265
```

STDIN is the predefined `filehandle` for standard input, i.e., the keyboard.

The line input operator is specified with `<filehandle>`

```
$new = <STDIN>;
```

If the input is a string value, we often want to trim off the trailing newline character (i.e. the \n), so we use the ubiquitous "chomp" function.

```
chomp($new = <STDIN>);
```

Equivalent to

```
$new = <STDIN>;
chomp($new);
```

File Handling

```
#!/usr/local/bin/perl
# basic09. pl COMP 519
# Program to open the password file,
# read it in, print it, and close it again.

$file = 'password.dat'; #Name the file
open(INFO, $file); #Open the file
@lines = <INFO>; #Read it into an array
close(INFO); #Close the file
print @lines; #Print the array
print $lines[0];   #  Print the first
                   #  line of the file
```

- Define filename `$file`
- `open` this file for input (reading) using filehandle INFO (or some other name)
- we can read the entire file into an array all at once using

```
@lines = <INFO>;
```

- `close` that file
- `print` the array (this will print the entire array in one go)

Can do even more:

```
open(INFO, $file);  #  Open for input, i.e. reading
open(INFO, "> $file");  #  Open for new output, i.e. (over)writing
open(INFO, ">> $file");   #  Open for appending
open(INFO, "< $file");    #  also for reading
```

Can use print with an extra parameter to print to a file, e.g.

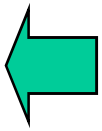
```
print INFO "This line goes to the file.\n";
```

(after opening the file for write/append operations)

Example

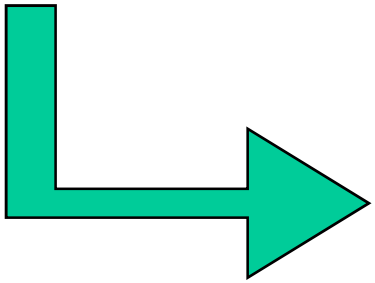
```
password.dat:

password1
password2
password3
password4
password5
...
...
...
```



```
#!/usr/local/bin/perl
# basic09. pl COMP 519
# Program to open the password file,
# read it in, print it, and close it again.

$file = 'password.dat'; #Name the file
open(INFO, $file); #Open the file
@lines = <INFO>; #Read it into an array
close(INFO); #Close the file
print @lines; #Print the array
```



```
bash-2.05b$ perl basic09.pl
password1
password2
password3
password4
password5
...
...
...
```

foreach-Statement

Can go through each line of an array or other list-like structure (such as lines in a file)

```
#!/usr/local/bin/perl
# basic10.pl COMP519
@food  = ("apples", "pears", "eels");
foreach $morsel ( @food )    # Visit each item in turn and call it
                             # $morsel
{
     print "$morsel \n";              # Print the item
     print "Yum yum\n";               # That was nice
}
```

```
bash-2.05b$ perl basic10.pl
apples
Yum yum
pears
Yum yum
eels
Yum yum
```

Testing

In Perl any non-zero number and any non-empty string is counted as **true**.

The number zero, zero by itself in a string, and the empty string are counted as **false**.

Here are some tests on numbers and strings.

```
$a == $b        # Is $a numerically equal to $b?
                # Beware: Don't use the = operator.
$a != $b        # Is $a numerically unequal to $b?

$a eq $b        # Is $a string-equal to $b?
$a ne $b        # Is $a string-unequal to $b?
```

You can also use logical and, or and not (typically as part of a conditional statement):

```
($a && $b)      # Are $a and $b both true?
($a || $b)      # Is either $a or $b true?
!($a)           # is $a false?
```

Control Operations

Operation	Numeric Operands	String Operands
Is equal to	==	eq
Is not equal to	!=	ne
Is less than	<	lt
Is greater than	>	gt
Is less than or equal to	<=	le
Is greater than or equal to	>=	ge
Compare, returning -1, 0, or +1	<=>	cmp

Loop Statements

for

```
#!/usr/local/bin/perl
# basic11.pl COMP519
for ($i = 0; $i < 10; ++$i)
# Start with $i = 1
# Do it while $i < 10
# Increment $i before repeating
{
     print "iteration $i\n";
}
```

Perl has loop statements that mimic those of C and Java

while

```
#!/usr/local/bin/perl
# basic12.pl COMP519
print "Password?\n”; # Ask for input
$a = <STDIN>; # Get input
chomp $a; # Remove the newline at end
while ($a ne "fred")
# While input is wrong..
{
  print "Sorry. Again?\n"; # Ask again
  $a = <STDIN>;  # Get input again
  chomp $a; # Chop off newline again
}
```

do-until

```
#!/usr/local/bin/perl
# basic13.pl COMP519
do
{
     print "Password? ";
     # Ask for input
     $a = <STDIN>; #Get input
     chomp $a; # Chop off newline
}
until($a eq "fred")
# Redo until correct input
```

Examples

for

```
bash-2.05b$ perl basic11.pl
iteration 0
iteration 1
iteration 2
iteration 3
iteration 4
iteration 5
iteration 6
iteration 7
iteration 8
iteration 9
```

while

```
bash-2.05b$ perl basic12.pl
Password?
wew
Sorry. Again?
wqqw
Sorry. Again?
fred
```

do-until

```
bash-2.05b$ perl basic13.pl
Password? werfwe
Password? qfvcc
Password? qcqc
Password? fred
```

Conditionals

Of course Perl also allows if/then/else statements.

```
if ($a)
{
 print "The string is not empty\n";
}
else
{
 print "The string is empty(ish)\n";
}
```

Remember that an empty string is FALSE. Also, “FALSE" if $a is the string “0” or ‘0’.

Notice, the elsif does have an "e" missing.

```
if (!$a) # The ! is the "not" operator
{
 print "The string is empty\n";
}
elsif (length($a) == 1)
# If above fails, try this
{
print "The string has one character\n";
}
elsif (length($a) == 2) # If that fails,
                        # try this
{
print "The string has two characters\n";
}
else # Now, everything has failed
{
print "The string has lots of
characters\n";
}
```

Strings and text processing

•One of the strengths of Perl is its capabilities for processing strings (and its great speed in doing so).

•Perl has many built-in functions for manipulating strings, some of which have been mentioned earlier, like (assuming that $str is a variable that holds a string)

```
$s = uc ($str);    #  Convert a string to uppercase
$l = length($str); #  Get the length of a string
```

•The real power comes from Perl's regular expression "engine" used for matching, substituting, and otherwise manipulating strings.

•Regular expressions exist in other languages (including Java, JavaScript, and PHP), but Perl has this built into the standard syntax of the language (where in other languages it's accessed through external libraries).

Regular Expressions

A regular expression (RE) is a template that matches, or doesn't match, a given string.

```
$sentence =~ /the/;
```
gives TRUE if the appears in $sentence

The RE is slashed / /, matching occurs because of the =~ operator (which is called the "binding operator"). So,

If `$sentence` is equal to `"The quick brown fox"` then the above match will be FALSE.

The RE is case sensitive. So,

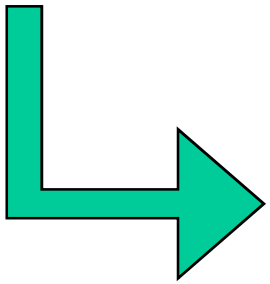
```
$sentence =~ /the/;
```
is FALSE because "the" does not appear in `$sentence`

The less-used operator !~ is used for spotting a non-match.

Example

```
#!/usr/local/bin/perl
# basic14.pl COMP519

$sentence = "The quick brown fox";
$a = ($sentence =~ /the/) ; $b= ($sentence !~ /the/);
print "Output gives $a and $b\n";
```



```
bash-2.05b$ perl basic14.pl

Output gives    and 1
```

Using the $_ Special Variable

Can use the Regular Expression as a conditional, e.g.

```
if ($sentence =~ /under/)
{
print "We're talking about rugby\n";
}
```

It works if we had either of

```
$sentence = "Up and under";
$sentence = "Best in Sunderland";
```

By assigning the sentence to the special variable $_ first, we can avoid using the match and non-match operators.

```
if (/under/)
{
print "We're talking about rugby\n";
}
```

Note: As mentioned previously, the $_ variable is the default variable for many Perl operations, and tends to be used very heavily.

More on REs

The creation of REs can be something of an art form.

```
.       # Any single character except a newline
^       # The beginning of the line or string
$       # The end of the line or string
*       # Zero or more of the last character
+       # One or more of the last character
?       # Zero or one of the last character
```

Here are some special RE characters and their meaning.

Some example matches:

```
t.e     # t followed by anything followed by e. This will match tre, tle but not te, tale
^f      # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters. This will match un, und, undd, unddd, ...
.*      # Any string without a newline. This is because the . matches anything except
                                    # a newline and the * means zero or more of these.
^$      # A line with nothing in it.
```

Remember that the RE should be enclosed in /.../ (slashes) to be used.

Even more on REs

Can use `[]` to match any one of the characters inside them.

```
[qjk]       # Either q or j or k
[^qjk]      # Neither q nor j nor k
[a-z]       # Anything from (lower-case) a to z inclusive
[^a-z]      # No lower case letters
[a-zA-Z]    # Any letter
[a-z]+      # Any non-zero sequence of lower case letters
```

a `-` indicates "between"

a `^` means "not":

Can use | for an "or" and (...) for grouping things together.

```
jelly|cream   # Either jelly or cream
(eg|le)gs     # Either eggs or legs
(da)+         # Either da or dada or dadada or...
```

Still More on REs

Can use some more special characters, and escape sequences

```
\n      # A newline
\t      # A tab
\w      # Any alphanumeric (word) character.
        # The same as [a-zA-Z0-9_]
\W      # Any non-word character.
        # The same as [^a-zA-Z0-9_]
\d      # Any digit. The same as [0-9]
\D      # Any non-digit. The same as [^0-9]
\s      # Any whitespace character: space,
        # tab, newline, etc
\S      # Any non-whitespace character
\b      # A word boundary, outside [] only
\B      # No word boundary
```

```
\|      # Vertical bar
\[      # An open square bracket
\)      # A closing parenthesis
\*      # An asterisk
\^      # A carat symbol
\/      # A slash
\\      # A backslash
```

* Find more on REs!

Some Example REs

It's probably best to build up your use of regular expressions slowly. Here are a few examples.

```
[01]          # Either "0" or "1"
\/0           # A division by zero: "/0"
\/ 0          # A division by zero with a space: "/ 0"
\/\s0         # A division by zero with a whitespace:
              # "/ 0" where the space may be a tab etc.
\/ *0         # A division by zero with possibly some
              # spaces: "/0" or "/ 0" or "/  0" etc.
\/\s*0        # A division by zero with possibly some whitespace.
\/\s*0\.0*    # As the previous one, but with decimal
              # point and maybe some 0s after it. Accepts
              # "/0." and "/0.0" and "/0.00" etc and
              # "/ 0." and "/  0.0" and "/   0.00" etc.
```

Remember that to use them for matching they should be put in /.../ slashes.

Substitution

As well as identifying REs, we can make substitutions using function s based on those matches.

```
$sentence =~ s/london/London/;
```

This replaces the first occurrence of london by London in the string $sentence

```
s/london/London/;
```

This does the same thing with the $_ variable.

```
s/london/London/g;
```

This makes a global substitution (in the $_ variable) by using the g option (for “global case") after the last slash.

This RE below can replace occurrences of lOndon, lonDON, LoNDoN and so on

```
s/[Ll][Oo][Nn][Dd][Oo][Nn]/London/g;
```

but an easier way is to use the i option (for "ignore case").

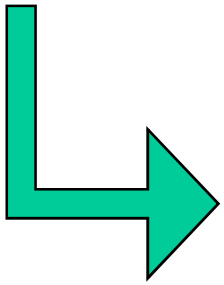
```
s/london/London/gi;
```

This will make a global substitution ignoring case.

The i option is also used in the basic /.../ regular expression match.

Example

```
#!/usr/local/bin/perl
# basic15.pl COMP519
$_ = "I love london, but not london or LoNdoN";
$a = s/london/London/;   # $a gives number of changes in $_
                         # but the changes are made to $_
print "$a changes in $_ \n";
$a= s/london/London/g;
print "$a changes in $_ \n";
$a= s/london/old London/gi;
print "$a changes in $_ \n";
```



```
bash-2.05b$ perl basic15.pl
1 changes in I love London, but not london or LoNdoN
1 changes in I love London, but not London or LoNdoN
3 changes in I love old London, but not old London or old London
```

Note that the changes themselves have occurred to the $_ variable, while $a stores the number of changes that have been made in each substitution.

Remembering Patterns

Can use the special RE codes `\1,\2,...` to recall matched patterns in the same regular expression or substitution.

```
$_ = "Lord Whopper of Fibbing";
s/([A-Z])/:\1:/g;
print "$_\n";
```

This will replace each upper case letter by that letter surrounded by colons, i.e.

`:L:ord :W:hopper of :F:ibbing.`

Can use the special Perl read-only variables `$1,$2,...` to recall matched patterns in the rest of the code (triggered by parentheses in the regular expression).

```
if (/(\b.+\b) \1/)
{
print "Found $1 repeated\n";
}
```

This will identify any words repeated within `$_`

```
$search = "the";
s/${search}re/xxx/;
```

This will replace every occurrence of `there` within `$_` with `xxx`.

After a match, you can use the special read-only variables `$`` and `$&` and `$'` to find what was matched before, during and after the search string.

Some Future Needs

```
s/\+/ /g;
s/%([0-9A-F][0-9A-F])/pack("c",hex($1))/ge;
```

`s/\+/ /g` globally substitutes all literal "+" signs, replacing them with a space.

`/%([0-9A-F][0-9A-F])/` means a % followed by two hexadecimal digits (the numerals 0-9 and the letters A through F).

() groups the two characters followed by the %, to later use it as $1.

`hex($1)` converts a hexadecimal number in $1 to a hexadecimal string.

`pack(template, list)` takes a list of values and packs it into a string according to the template given.
The "c" tells it to put the hexadecimal number into a character.
e evaluate the right side as an expression.

Translation

The `tr` function allows character-by-character translation.

```
$sentence =~ tr/abc/edf/
```

This replaces each a with e, each b with d, and each c with f in the variable $sentence, and returns the number of substitutions made.

Most of the special RE codes do not apply in the `tr` function.

```
$count = ($sentence =~ tr/*/*/);
```

This counts the number of asterisks in the `$sentence` variable and stores that in the `$count` variable.

However, the dash is still used to mean "between".

So consider these examples below:

```
tr/a-z/A-Z/;
```

This converts `$_` to upper case.

```
tr/\,\.//;
```

This deletes all commas and periods from `$_`

Split

The `split` function splits up a string and places it into an array.

```
$info = "Caine:Michael:Actor:14 Leafy Drive";
@personal = split(/:/, $info);
```

```
@personal = ("Caine", "Michael",
"Actor", "14 Leafy Drive");
```

```
@personal = split(/:/);
```

Operates on the variable `$_`

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:/);
```

```
@personal = ("Capes", "Geoff",
"",  "Shot putter", "",
"", "Big Avenue");
```

Can use REs inside of the split function

```
$_ = "Capes:Geoff::Shot putter:::Big Avenue";
@personal = split(/:+/);
```

```
@personal = ("Capes", "Geoff",
"Shot putter", "Big Avenue");
```

```
@chars = split(//, $word);
@words = split( /, $sentence);
@sentences = split(/\./, $paragraph)
```

A word into characters
A sentence into words
A paragraph into sentences

Some Future Needs

```
# split string on '&' characters
@parameter_list = split(/&/,$result);

foreach (@parameter_list) {
    s/\+/ /g;
    s/%([0-9A-F][0-9][A-F])/pack("c",hex($1))/ge;
}
```

Firstly, we chop the string up, dividing it on each `&`, and placing the results in an array of strings called `@parameter_list`.

Secondly, we loop over the array. Each element in the array is, in turn, implicitly assigned to `$_` (the default variable), and the substitution commands in the loop are applied to `$_`, (modifying the value of that variable in the parameter list).

Hashes

Hashes (associative arrays) are arrays in which each data element is accessed by a string key.

```
%ages = ("Michael Caine", 39,
         "Dirty Den", 34,
         "Angie", 27,
         "Willy", "21 in dog years",
         "The Queen Mother", 108);
```

```
$ages{"Michael Caine"}; # 39
$ages{"Dirty Den"};  # 34
$ages{"Angie"}; # 27
$ages{"Willy"};  # "21 in dog years"
$ages{"The Queen Mother"}; # 108
```

On the right (when accessing one element of the hash) we see that each % has changed to a $ (as each individual hash element is a scalar). The index is enclosed in curly braces.

Can be converted back into a list array

```
@info = %ages;          # @info is a list array. It now has 10 elements
$info[5];               # Returns the value 27 from the list array @info
%moreages = @info;      # %moreages is a hash.
                        # It is the same as %ages
```

Operators

Can access all the elements of a hash in turn using the keys function and the values function

```
foreach $person (keys %ages)
{
print "I know the age of $person\n";
}
foreach $age (values %ages)
{
print "Somebody is $age\n";
}
```

`keys` returns a list of the keys (indices)
`values` returns a list of the values.
In which order? (No guaranteed order.)
If called in a scalar, return the number of key/value pairs in the hash.

```
while (($person, $age) = each(%ages))
{
print "$person is $age\n";
}
```

`each` returns a two element list of a key and its value. Every time `each` is called it returns another key/value pair.

```
if (exists $ages{"The Queen Mother"})
{
print "She is still alive…\n";
}
```

exists returns TRUE if the value exists as a *key* in the hash.

```
delete $ages{"The Queen Mother"};
```

delete deletes the specified key and its associated value from the hash.

Subroutines

Can define and place own functions (subroutines) anywhere (ok, nearly anywhere) in code.

```
sub mysubroutine
{
print "Not a very interesting routine\n";
print "This does the same thing every time\n";
}
```

A user-defined subroutine is called with an & character in front of the name.

```
&mysubroutine;                  # Call the subroutine
&mysubroutine($_);              # Call it with a parameter
&mysubroutine(1+2, $_);         # Call it with two parameters
```

A function call isn't type-checked. It may be called with none, one, or several parameters, regardless of how it is defined.

Parameters

When the subroutine is called any parameters are passed as a list (array) in the special `@_` array variable. Individual elements of this array are accessed as $_[0], $_[1], etc.

```
sub printargs
{
print "@_\n";
}

&printargs("perly", "king");        # Example prints "perly king"
&printargs("frog", "and", "toad"); # Prints "frog and toad"
```

Just like any other list array the individual elements of `@_` can be accessed with the square bracket notation.

```
sub printfirsttwo
{
print "Your first argument was $_[0]\n";
print "and $_[1] was your second\n";
}
```

The indexed scalars `$_[0]` and `$_[1]` (and so on) in the subroutine have nothing to with the global scalar `$_` which can also be used without fear of a clash.

Returning Values

Subroutines can return a value, and may use the return keyword (but this is not always required).

```
sub maximum
{
  if ($_[0] > $_[1])
  {
  return $_[0];
  }
  else
  {
  return $_[1];
  }
}
```

```
$biggest = &maximum(37, 24);

# Now $biggest is 37
```

The return value of a function is always the last thing evaluated.

```
sub printfirsttwo
{
print "Your first argument was $_[0]\n";
print "and $_[1] was your second\n";
}
```

What is the return value of

$printfirsttwo(37,24); ????

It's actually undefined in this case…

Local Variables

The `@_` variable is local, and so, of course, are `$_[0], $_[1], $_[2],` and so on (when used inside the subroutine). Other variables can be made local too.

By default, variables are global in Perl (so there is a danger of altering variable values outside of the subroutine if you're not careful).

```
sub inside
{
my ($a, $b);                    # Make local variables using the
                                #      "my" keyword
($a, $b) = ($_[0], $_[1]);      # Assign values
$a =~ s/ //g;                   # Strip spaces from
$b =~ s/ //g;                   # local variables
($a =~ /$b/ || $b =~ /$a/);     # Is $b inside (contained in) $a
                                #    or $a inside $b?
}

&inside("lemon", "dole money"); # true
```

In fact, this subroutine can even be tidied up by replacing the first two lines with

```
my ($a, $b) = ($_[0], $_[1]);
```

More on Parameters

- The number and type of parameters for a Perl subroutine are not necessarily fixed.
- It is up to the programmer to ensure that the parameter types are correct and/or that the parameters make sense in the context they are used.
- Variable length parameter lists allow you to write subroutines of this type:

```
sub maximum
{   # returns the maximum of the
    #  parameters passed to function
    my ($current_max) = shift @_;

    foreach (@_)
        {
          if ($_ > $current_max)
               {
                 $current_max = $_;
               }
        }
    $current_max;   #  return the max
}
```

This subroutine uses the default (local) variable $_ to store each element in the "foreach" statement as it "loops" through the array elements.

References

A reference is a scalar that references another variable or a literal.

```
$age=42;
$ref_age =\$age;
@stooges = ("Curly","Larry","Moe");
$ref_stooges = \@stooges;
$ref_salaries = [42500,29800,50000,35250];
$ref_ages ={
            'Curly' => 41,
            'Larry' => 48,
            'Moe' => 43,
};
```

For referencing,

use \ on the name of the variable,

[] for arrays

{} for hashes

```
$$ref_age = 30;
$$ref_stooges[3] = "Maxine";
$ref_stooges -> [3] = "Maxine";
```

For dereferencing,

use $$ on the reference name, or

-> for arrays and hashes

*Find more on references!

Sorting

- Perl has a built-in sorting function, called, as one might expect “sort”.
- Using this function is usually straightforward enough, as in this example:

```
@names = ("Joe", "Bob", "Bill", "Mark");

@sorted = sort @names;   #  @sorted now has the list
                         # ("Bill", "Bob", "Joe", "Mark")
```

- You must be careful using this function with numbers, though, as the default will sort them into the ASCII-betical order.
- To get them into the proper numerical order, you must supply the sort function with a comparison routine. One way to do this is the following:

```
@numbers = (12, 3, 1, 8, 20);

@sorted = sort { $a <=> $b } @numbers;
                             #  @sorted now has the list
                             # (1, 3, 8, 12, 20)
```

- This comparison routine tells Perl how to compare the two values using the built-in “spaceship” (“TIE fighter”?) operator.
- You can also sort in the reverse order by using the keyword “reverse”.

```
@numbers = (12, 3, 1, 8, 20);
@sorted = reverse sort { $a <=> $b } @numbers;     #  sort the list from biggest
                                                   #    to smallest
```

What is Next?

In the next lecture we move to CGI programming.

We will learn how Perl can be used for creating CGI programs that can take user input, process it, and return a new web page to the user.

We'll see how to access the environmental variables which contain a lot of information, how to access information that a client submits to the server, and how to create a new web page using Perl.