Perl — A Quick Review of the Basics

by

Avi Kak
kak@purdue.edu

January 17, 2006

This review was written specially for the students taking ECE 495F, Introduction to Computer Security, to help them with writing Perl scripts for solving homework problems.

- Perl is interpreted — but not literally so because there is a compilation stage that a Perl script goes through before it is executed.

- The executable *perl*, normally installed in `/usr/bin/` or `/usr/local/bin`, when invoked on a script first converts it into a parse tree, which is then translated into a bytecode file.

- It is the bytecode that is interpreted by *perl*. The process of generating the bytecode is compilation.

- This compilation phase goes through many of the same optimization steps as the compilation of, say, a C program — such as eliminating unreachable code, replacing constant expressions by their values, loading in library definitions for the operators and the builtin functions, etc.

- For those who are steeped in the highly-structured mainstream programming languages like C, C++, Java, etc., perhaps the biggest first surprise associated with Perl is that it is typeless.

- By that we mean that a variable in Perl does not have to be declared as being of any specific type before it is initialized.

- A variable that was assigned a number previously can be assigned a string in the next statement and vice versa.

Scalars, Arrays, and Hashes

- In Perl, a variable may hold a single value, in which case it is called a *scalar variable*; or a variable may hold a list of values, in which case it is called an *array variable*; or it may hold a collection of (key, value) pairs, in which case it is called a *hash variable*.

- A scalar variable may be referred to as a *scalar* for short.

- Similarly, an array variable may be referred to as an *array* for short.

- And a hash variable as a *hash* for short.

- Scalars, arrays, and hashes may also be thought of as Perl's built-in *containers*.

- Obviously, a scalar can only hold a single value, often referred to as a *scalar value*; an array a sequence of scalar values; and a hash a collection of (key, value) pairs where keys and values are scalar values.

Scalar

- A scalar variable's value can be a string, a number, or a reference.

- The prefix '`$`' on the variables `$x` and `$y` in statements such as

  ```
  $x = 4;
  $y = "hello";
  ```

  means that these variables are scalars. More precisely, the prefix '`$`' stands for the fact that *we expect the variable to return a scalar value.*

- A scalar can be *interpolated* into a double-quoted string.

- Interpolation means that the print representation of the value that the scalar variable is holding will be substituted at the location of the scalar variable.

- For example, we can say

  ```
  $x = "Trillian";
  $y = "Good morning to $x";
  ```

  This will cause the the value of `$y` to become "`Good morning to Trillian`". **Note that Perl comes with two types of strings: double-quoted and single-quoted. Only the double-quoted strings can be interpolated into.**

Array

- An array is a collection of scalars.

- The name of an array variable must be prefixed by the symbol '`@`', as in the following statement that defines an array named `@arr` that contains the four scalar values shown:

```
@arr = (1, 2, 3, "hello");
```

  What we have on the right of the above assignment is a *list literal*, in this case consisting of the four comma-separated scalar values shown.

- A *list* in general is an ordered list of scalar values and an *array* is a variable whose value is a list.

- Since an array element — which is the same thing as the element of the list that is the value of the array variable — is always a scalar value, it can be accessed through an index in the following manner:

```
print "$arr[1]\n";              # will output 2 and then a r

$arr[3] = "jello";              # change the fourth element
```

  The important thing to note here is that since we expect an array element to be a scalar value, we access it with the same '`$`' prefix that we use for accessing the value of any ordinary scalar variable.

- The list that is used to initialize an array is allowed to contain expressions for its elements. When that happens, each such expression is evaluated in what is known as the *list context* and the result returned by such evaluation substituted into the list.

- When Perl executes the second statement below, the array `@friends` is evaluated in a list context. This evaluation will return the list of scalar elements shown on the right in the first statement.

- Each element returned by the evaluation of `@friends` is *inserted* into the list on the right of the assignment operator in the second statement.

```
@friends = qw/ harry sally monica joe /;
@arr = (1, 2, 3, @friends, 4);
```

  As a result, the list assigned to `@arr` will be `(1, 2, 3, "harry", "sally", "monica", "joe")`.

- The operator `qw` causes its string argument to be broken into a list of words with the intervening white space as word delimiters. It returns the list of words so formed. *This list is formed at compile time.*

- An important aspect of Perl arrays is that it is not an error to access an element outside the range dictated by the list used for initializing an array variable. Such an access will simply return Perl's null object `undef`.

- It is also not an error to assign a value to an array element whose index is outside the range specified by the list used for initialization.

- Like all variables in Perl, an array element, no matter what its index, will spring into existence simply by assignment.

- **This property of Perl — array and hash elements coming into existence through assignment — is known as *autovivification.*** When that happens, each intervening array element, if previously nonexistent, is given the value `undef`.

- A small array is usually displayed by interpolating it directly into a string as follows:

```
@arr = ('one', 'two', 'three', 'four');
print "@arr\n";                 # one two three four
```

  The interpolation causes the array variable to be evaluated in a list context.

- However, when `@arr` is evaluated in a scalar context, it returns an integer whose value is the number of elements in the array.

Hash

- Another important built-in data type in Perl is *hash* for storing associative data when it comes in the form of key-value pairs.

- As shown below, a list can be used to initialize a hash variable with the key-value pairs:

```
%phone_book = ( zaphod          =>  3456789,
                beetlegeuse     =>  0123456,
                trillian        =>  0001111 );
```

  where on the left, by virtue of the prefix '%', we have declared `%phone_book` to be a hash variable.

- On the right we have a list literal, although this time it is shown with the big arrow ('`=>`') between alternate elements to make it easier to see the keys and their associated values.

- Use of the big arrow in the manner shown also makes it possible to display the keys as *barewords*.

- If we had used a list literal with all elements separated by commas, we'd have to quote the keys that must always be strings. The big arrow implicitly quotes the keys on the left of the big arrow.

- Only strings, or expressions that evaluate to strings, can be used for hash keys and there can only be one key corresponding to a given string. In other words, all the keys in a hash are all strings and they are all distinct and unique.

- A value, on the other hand, can be any scalar, and a given value can appear any number of times in a hash.

- To be more specific, a value for a given key could be a number, a string, or a reference. (We will talk about Perl references later.)

- The scalar value for a given key in a hash can be accessed by using the '`$`' prefix, as for any ordinary scalar, and by enclosing the key in curly braces. For example, we would invoke

  ```
  $phone_book{ zaphod }
  ```

  to retrieve the value associated with the key "zaphod" in the hash `phone_book`.

- If we wanted to change the value associated with this key, we could say

  ```
  $phone_book{ zaphod } = 7893456;
  ```

- The syntax for fetching a hash value can be used directly for interpolating such values in a double quoted string. For example, we could say

```
$key = "zaphod";
print "$key's phone number is $phone_number{$key}\n";
```

- Perl gives us two functions, `keys()` and `values()`, that when invoked on a hash in list contexts return, respectively, a list of all the keys and a list of all the values. For example, with the hash `%phone_book` declared and initialized as shown previously, if we say

  ```
  @k = keys %phone_book;
  @v = values %phone_book;
  ```

  the list of keys returned by the `keys()` function will be placed in the array `@k` and the list of values returned by `values()` in the array `@v`.

- The keys and values are returned in the order, apparently random, in which they happen to be stored in the hash.

- Each operation on a hash that either inserts a new key or deletes an existing key is potentially capable of changing the order in which the elements of a hash are stored.

- The keys and the values that are returned by the operators `keys` and `values` are copies of the actual items contained in the hash. So modifying one or more elements of the arrays `@k` and `@v` will not change the contents of the hash `%phone_book`.

- When the functions `keys()` and `values()` are invoked in a scalar context, you simply get the number of elements — meaning the number of (key, value) pairs — in a hash.

- For the same `%phone_book` hash we have used in this section, if we say

```
$how_many = keys %phone_book;                    # 3
$how_many = values %phone_book;                  # 3
```

  the scalar `$how_many` would be set to the number 3 in each case.

- The fact that, in a scalar context, the `keys` operator applied to a hash returns a scalar can be used to set the number of buckets pre-allocated to the hash.

- Without such pre-allocation, the number of buckets will increase in a more-or-less incremental fashion as new entries are made into a hash.

- While a new (key, value) pair is inserted into a hash merely by assignment, as in

```
$phone_book{ arther_denton } = 34521;
```

  an existing (key, value) pair may be removed from a hash by using the `delete()` function:

```
delete $phone_book{ arthur_denton };
```

- Whether or not a (key, value) pair for a given key exists in a hash can be ascertained by invoking the function `exists()` on the key, as in

  ```
  exists $phone_book{ arther_denton }
  ```

  This invocation will return true if the hash `phone_book` contains a (key, value) pair for the "arther_denton" key.

- It is frequently the case that one wants to scan a hash, element by element. **This is also referred to as *iterating* over a hash.** The function `each()` when invoked on a hash will return the next (key, value) pair. So we could use `each()` in a `while` loop in the following manner

  ```
  while ( ($name, $number ) = each %phone_book ) {
      print "$name => $number\n";
  }
  ```

- Another way to iterate over a hash is to first fetch all the keys by using the `keys()` function in the manner already described and then using the `foreach` control structure to iterate over the keys.

Lexical versus Global Names

- Once a variable is defined, the important question is as to where it will be visible in the rest of the script.

- The basic distinction to be made here is between *lexical variables* and *global variables*.

- From the point of its definition, a lexical variable is visible only within the lexical scope in which it is defined. There is simply no way to access such a variable outside the lexical scope that contains its definition.

- A global variable, on the other hand, can be accessed anywhere within a script.

- It helps to make a distinction between two different kinds of global variables: the *package variables* and the system-supplied truly global variables.

- A package variable can be accessed anywhere in a script after its declaration, especially provided its *package-qualified* name is used for that purpose.

- On other hand, a system-supplied variable, such as the Perl's default variable '`$_`', can be accessed anywhere in a script without any special prefixes or anything.

Lexical Variables

- A lexical variable is defined with a `my` declaration, as in

```
my $x = 4;
my $y = "hello";
my @arr = (1, 2, 3, "four");
my %phone_book = undef;
```

- Lexical variables have *lexical scope*.

- What that means is that such variables are visible from the point of their declaration to the end of the lexical scope.

- A lexical scope (also known as *static scope*) will generally correspond to a curly brace delimited block of code.

- But it can also correspond to the string argument given to the `eval` operator.

- For lexical variables declared at the beginning of a file outside any code block, the lexical scope will cover the entire file.

- Lexical variables of the same name may be defined in *nested scopes*, as we do with the variable '`$x`' in the following scriplet:

```
my $x = 2;
```

```
my $y = 100;
print "$x, $y \n";                          # 2 100
{
    my $x = 4;
    print "$x, $y \n";                      # 4 100
    {
        my $x = 8;
        print "$x, $y \n";                  # 8 100
    }
    print "$x, $y \n";                      # 4 100
}
print "$x, $y \n";                          # 2 100
```

What this shows is that an outer lexical scope covers all inner lexical scopes and that a declaration for a variable in an inner scope hides its outer scope definition. *Upon exit from the inner scope, the outer scope definition of such a variable is restored.*

- Multiple variables may simultaneously be declared to be lexical by placing them all in parentheses, as in

```
my ($x, @arr, %phone_list);
$x = 4;
@arr = (1, 2, 3, "four");
%phone_list = ( peter => 123, paul => 234, mary => 345 );
```

Package Variables

- Whereas a lexical variable is visible only inside its own lexical scope, a package variable is visible everywhere after the point of its declaration.

- Package variables, brought into existence either with an `our` declaration or with package-qualified naming syntax, are stored in a namespace dictionary separately for each package.

- The thread of execution of a Perl script is placed in a *package namespace* by invoking the `package` declaration.

- Once the execution starts in a namespace, it continues in that namespace until another `package` declaration is seen.

- However, if a `package` declaration is within a separate lexical scope, the thread of execution would revert to the namespace of the enclosing lexical scope at the end of the inner lexical scope.

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

# PackageNames.pl

use strict;

my $a = 10;                                                    #(A)
$main::x = 100;                                                #(B)
our $y = 101;                                                  #(C)
print "$a \n";                              # 10               #(D)
#print "$x \n";                             # ERROR            #(E)
print "$y \n";                              # 101              #(F)
print "$main::x \n";                        # 100              #(G)
print "$main::y \n";                        # 101              #(H)

package Pack1;                                                 #(I)
my $b = 20;                                                    #(J)
our $x = 1000;                                                 #(K)
$Pack1::y = 1001;                                              #(L)
print "$x $y \n";                           # 1000 101         #(M)
print "$x $Pack1::y \n";                    # 1000 1001        #(N)


package Pack2;                                                 #(O)
my $c = 30;                                                    #(P)
our $x = 2000;                                                 #(Q)


package Pack1;                                                 #(R)
{                                                              #(S)
    package Pack3;                                             #(T)
    our $x = 3000;                                             #(U)
    $Pack3::y = 3001;                                          #(V)
    $Pack2::y = 2001;                                          #(W)
```

```
    $Pack3::z = 3002;                                                    #(X)
}                                                                        #(Y)
our $z = 1002;                                                           #(Z)

package Pack2;                                                           #(a)
our $z = 2002;                                                           #(b)

package main;                                                            #(c)
our $z = 102;                                                            #(d)


print "$a $b $c \n";                              # 10 20 30            #(e)
print "$x \n";                                    # 2000                #(f)
print "$main::x $main::y $::z \n";                # 100 101 102         #(g)
print "$Pack1::x $Pack1::y $Pack1::z \n";         # 1001 1002 1003      #(h)
print "$Pack2::x $Pack2::y $Pack2::z \n";         # 2001 2002 2003      #(i)
print "$Pack3::x $Pack3::y $Pack3::z \n";         # 3001 3002 3003      #(j)


print join "\n", keys %Pack1::;                                          #(k)
                                                  # x
                                                  # y
                                                  # z
print join "\n", keys %main::;                                           #(l)
                                                  # STDOUT
                                                  # STDIN
                                                  #  ...
                                                  # x
                                                  # y
                                                  # z
                                                  #  ...
```

Pragmas

- The '-w' switch in the first line of the script shown on page 17 turns on a large number of compiler warnings like accessing an uninitialized variable, finding a string where a number is expected, redefining a subroutine, attempting to write to a stream that is opened only for reading, etc.

- These warnings can also be turned on by invoking the *warning pragma* by

  ```
  use warnings;
  ```

  and turned off by

  ```
  no warnings;
  ```

- Pragma declarations have *lexical scope*, meaning that when warnings are turned off or on, that condition applies from the point of the declaration to the end of the lexical scope in which the pragma is invoked.

- After exiting from the scope, the warnings will be treated as specified by the outer enclosing block.

- The third line of the script shown on page 17 makes the *pragma* declaration

  ```
  use strict;
  ```

This pragma is a shorthand for the following three strictures:

```
use strict "refs";
use strict "vars";
use strict "subs";
```

- The first of these — `use strict "refs"` — triggers a compile-time error if a script uses a symbolic reference.

- The second — `use strict "vars"` — evokes an error from the compiler if you access a non-`my` variable without using a package-qualified name for the variable, unless it is one of Perl's predefined variables (such as `$_`, `@ARGV`, etc.) or a variable specifically imported from another package.

- The last of the three pragma declarations shown above — `use strict "subs"` — disallows the use of barewords, meaning unquoted strings, in a script unless the meaning of such words can be inferred from the script processed up to that point or such a word is used as a key in a hash.

- As with the `use warnings`, all of the `strict` pragmas have lexical scope.

- The `use strict` pragma can be turned off by the declaration `no strict`.

Control Structures

- Perl has the `if` and `unless` control structures for conditional evaluation of a block of code.

- The `if` control structures works in the same way as it does in all computer languages — a block of code is evaluated subject to what is known as a *conditional* or a *controlling expression* being true. The `unless` control structure works in a reverse fashion — a block of code is evaluated subject to the conditional of the `unless` construct being false.

- Perl also gives us the `while`, `until`, `foreach`, and `for` control structures for setting up loops for iterative (but conditional) evaluation of a block of code.

- Of these control structures, whereas the `while` control structure continues to loop through a block of code as long as the conditional of the `while` construct is true, the `until` control structure continues to loop through a block of code as long as the conditional is false.

- With regard to the Perl's control structures for loops, it is important to understand the role played by the default variable `$_` that Perl may use implicitly as the *loop control variable* in the absence of a user-defined loop control variable.

```
#!/usr/bin/perl -w                                          #(A)

## While_If.pl

use strict;                                                 #(B)

print "Enter numbers, one per line.".
             "When done, just enter return\n";              #(C)
my $sum;                                                    #(D)

while (<>) {                                                #(E)
    if (/^\s*$/) {                                          #(F)
        printf "Sum of numbers is: %d\n", $sum;             #(G)
        last;                                               #(H)
    }                                                       #(I)
    $sum += $_;                                             #(J)
}
```

- Another looping mechanism in Perl, provided by the `foreach` control structure, is convenient for walking through an array, with the *loop control variable* set to each element of the array, and doing something depending on the value of that element.

- This is illustrated by the following example where the `qw` operator at the right hand side of line (A) returns a list obtained by quoting each item in the argument supplied to `qw`.

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

```
## foreach.pl

use strict;

my @files = qw{/home/kak/perl/perl-run
               /home/kak/perl/perl-functions};     #(A)

foreach (@files) {                                 #(B)
    print "$_ looks like a text file and " .
                "is readable\n" if -r && -T;       #(C)
}
```

- Now how by placing the items of the argument to the `qw` operator in two different lines, line (A) also shows that you can use white space as needed in order to make your program visually easier to read.

- The `foreach` control structure in line (B) uses the Perl's default variable `$_` as the loop control variable. Perl sets `$_` to each element of the list argument supplied to `foreach` and then processes the block of code that follows.

- In the example shown, for the first iteration, the variable `$_` will be set to the file name shown in the first argument to `qw`. The statement in line (C) will then apply the file tests `-r` and `-T` to this file. The former test checks whether the permission bits set

for the file make it readable by whosoever is running the Perl program and the latter checks whether the file is a text file.

- Also note in line (C) the backward form of the `if` statement. What follows `if` is the condition that must evaluate to true for what comes before `if` to be executed.

- As mentioned at the beginning of this section, the control structure `unless` reverses the sense in which the condition must succeed in an `if` statement, and the loop control structure `until` plays the same role vis-a-vis `while`.

- The following example program illustrates both `until` for setting up a loop and `unless` for conditional evaluation of a statement.

- The goal of the program to report on the occurrence of a specified string within a larger string. The larger string is supplied in lines (A) and (B). Note the use of the dot operator '.' in line (A) to join quoted string in line (A) with the quoted string in line (B). Line (C) specifies the substring whose occurrence we want the program to search for in the string supplied through lines (A) and (B).

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

## until_unless.pl
```

```
use strict;

my $string = "The difference between reality ".
                "and fiction?  The fiction has " .
                "to make sense";                        #(B)
my $str = "fiction";                                    #(C)

my $where = 0;                                          #(D)
until ( $where == -1 ) {                                #(E)
    $where = index( $string, $str, $where );            #(F)
    unless ( $where == -1 ) {                           #(G)
        print "$where\n";                               #(H)
        $where++                                        #(I)
    };
}
```

- The actual substring search in the script is carried out by the `index` function call in line (F). This function returns '-1' if the substring supplied via the second argument is not found in the string supplied via the first argument. If the substring is found, the value returned is the zero-based index of the first character in the long string that matches the first character of the substring. The search starts at the position indexed by the value supplied through the last argument.

- That brings us to the `for` control structure, which works just like the `for` in C, as shown by the following example:

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

## for.pl

use strict;

for (my $i = 0; $i < 10; $i++) {
    printf "square of $i is %d\n", $i * $i;
}
```

which produces the following sort of output:

```
square of 0 is 0
square of 1 is 1
square of 2 is 4
 ....
 ....
```

The above example also illustrates the `printf` function for formatted output. It works like C's `printf` function.

- It is also possible to use `for` as an abbreviation for `foreach`. What is shown below is really a `foreach` loop written with `for`:

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

## for2.pl

use strict;
```

```
for (0..9) {                                    #(A)
    printf "square of $_ is %d\n", $_ * $_;      #(B)
}
```

This example also shows the range operator '..' in line (A) which creates a list of values starting from the left scalar, incrementing it repeatedly by one, until the value equals the right scalar. So the construct `(0..9)` returns the list '(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)'. Just as would be the case for a `foreach` loop, Perl's default variable `$_` is used for loop control in the above example. This variable is set successively to the values in the list (0, 1, ..., 9).

- Perl, being typeless, does not have a Boolean type. Instead, Perl examines the value returned by the conditional and makes a decision as to whether it is true or false using the following criteria:
  - The special value `undef` is false. When a scalar variable is declared in Perl, but left uninitialized, it is given a special value named `undef`. On the other hand, when an array is declared, but left initialized by the programmer, it is initialized to the empty list '()'.
  - The numbers 0 and 0.0 are false. All other numbers are true.
  - The empty strings '' and "" are false. All other strings are true save for the two exceptions '0' and "0" that are also interpreted as false.

- The examples we have shown so far for the conditionals in the

various control structures used simple expressions.

- As with other languages, one can also use more complex conditions in the control structures. These would be *logical expressions* built from the *numeric relational operators* (¡, ¡=, ¿ and ¿=), the *numeric equality operators* (== and !=), the *string relational operators* (lt, le, gt, ge) the *string equality operators* (eq and neq), the *binary logical operators* (`&&`, `||`) and the unary logical negation operator (!).

- When a logical expression in a condition has two or more Boolean elements connected by either the `and` operator `&&` or the `or` operator `||`, Perl carries out a *short-circuit evaluation* of the logical expression. Given a logical expression of the form

  ```
  A && B
  ```

  short-circuit evaluation means that if `A` is evaluated to be false, Perl will ignore B since the Boolean value of the entire logical expression will be false anyway. Similarly, in a logical expression of the form

  ```
  A || B
  ```

  if `A` evaluates to true, Perl will ignore `B` since the Boolean value of the entire expression will be true no matter what the value of `B`.

- When a logical expression `A && B` is true or when `A || B` is false

(meaning when both the operands are evaluated), it is the value of the right operand that is returned.

- This can be useful for assigning default values to variables, as in the example below.

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

## default_with_or.pl

use strict;

die "Too many command line args\n" unless @ARGV <= 5;  #(A)

my $sum;

foreach ( 1..5 ) {                                     #(B)
    my $entry = shift @ARGV || -10;                     #(C)
    $sum += $entry;                                     #(D)
}

my $avg = $sum / 5.0;                                   #(E)

print "The average over five entries is $avg\n";
```

- Also note in line (A) of the above example our first demonstration of the very commonly used `die` function to terminate a program should certain conditions not be fulfilled. The `die` function

sends to the standard error stream, which would ordinarily be your computer terminal, the message that is supplied to it as the argument.

- While we are on the subject of the `die` function, note that this function exhibits a slightly different behavior depending on whether or not you include the newline character at the end of the argument string.

- In line (A) of the previous program, we included the newline terminator in the message to be displayed. This causes just this message to be printed out if the program is to be terminated.

- However, if we had not included the trailing newline, the message sent to the standard error stream would also include at the end the program name and the line number that caused the termination of the program.

- Another variation on the use of `die` is exemplified by the following syntax where we want the program to be terminated if a system call fails for some reason:

```
open( FILE, "< data.txt" )  ||
    die "Unable to open the file: $!";
```

- Perl also allows you to use the keyword `and` for the `&&` operator and the keyword `or` for the `||` operator.

- Although the operators `and` and `or` are logically identical to the operators `&&` and `||`, respectively, they differ in one very important manner: the precedence levels of `and` and `or` are much lower compared to the precedence levels of `\&\&` and `||`.

- As a result, the `and` and the `or` operators do not stick to their operands as tightly as the `&&` operators, making for a reduced need for parentheses for the grouping of the operands. The following program illustrates the use of `or` in line (A):

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

## or.pl

use strict;

open( FILE, "< data.txt" )
             or die "Unable to open file: $!";       #(A)

local $/ = undef;                                    #(B)

my @all_words = split /\s+/, <FILE>;                 #(C)

my $num_of_words = @all_words;                       #(D)

print "Number of words in the text file: $num_of_words\n";
                                                     #(E)
close FILE;                                          #(F)
```

- There is another interesting Perl feature illustrated in this program. Line (B) illustrates the use of `local` declaration for a language-supplied global variable. When such a global variable is declared `local`, its value is temporarily put away until the end of the scope of the `local` declaration.