Perl Primer

An Introduction to Perl for C++ Programmers

by Frank McCown and Tim Baird
Harding University

PERL is the Practical Extraction and Report Language (or Pathologically Eclectic Rubbish Lister)

- Developed by Larry Wall who is still the chief architect.
- Part of the Open Source software movement- distributed under the GNU Public License.
- Available for many operating systems including Linux, Unix, Win 2000/XP, VMS
- May be retrieved from the CPAN (Comprehensive Perl Archive Network) http://www.perl.com/CPAN
- Free Windows version can be obtained from ActiveState's site http://www.activestate.com
- Frequently asked questions: http://language.perl.com/faqo or manual pages (`$man perl`)

Perl has two major uses:

1) general purpose scripting language for system administration
   - similar, but more powerful than Unix shell scripts, VMS DCL scripts, DOS batch scripts
   - has very powerful regular expression operators
   - has extremely easy-to-use file and directory manipulation functions
   - supercedes AWK and SED utilities
   - can be used in ASP as server-side scripting language
2) most CGI programming
   - has a nice library for standard CGI parsing
   - has been called the “duct tape” of the Web

Perl is an interpreted language:

- There are two passes, so all syntax errors are found before execution starts.
- The first pass produces a binary byte-code, but not machine language.
- Speed is considered to be excellent even though it is interpreted.
- The interpreter lends itself to interactive experimentation.

Perl is used differently depending on the operating system:

- On Linux:

  Create a text file for the script and give it executable permissions. Make sure the first line is: #!/usr/bin/perl
  Invoke it at the $ prompt by typing: `$`*`filename`*

- On Windows:
  Create a text file for the script. Invoke it at the DOS prompt by typing: `C:>perl` *`filename`*

Hello, *Name*! Example:

```
#!/usr/bin/perl

print "What is your name? ";
$name = <STDIN>;
chomp($name);

if ($name eq "McCown") {
  print "Hello, Teacher!\n";
}
else {
  print "Hello, $name!\n";
}
```

Hello, World! CGI Example:

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<HTML><BODY>Hello, World!</BODY></HTML>";
```

Place hello.cgi file in /home/*username*/public_html/cgi-bin directory. Make sure hello.cgi has user read and execute permissions.

Access from browser at:
http://taz.harding.edu/cgi-bin/cgiwrap/*username*/hello.cgi

Table of Contents

I. Comments - # through end of line. No multi-line comments.

II. Variables and Data Types

A. Variables do not have to be initialized before using. If a variable is referenced before it is assigned a value, it will yield 0 or the empty string (“”).

B. Variable names begin with special char ($, @, or %), a letter, then letters, digits, and underscores. Max of 255 characters, case sensitive.

C. Variables are scalar, arrays, or associative arrays (hashes)

1. Scalars begin with $: `$hits = 1;`
2. Arrays begin with @: `@numbers = (1, 3, 10);`
3. Hashes begin with %. `%employees = (456 => "Sue", 762 => "Jack");`

D. No char variables, just use string

E. No int variables, numbers always stored as floats

F. Variables are global by default.

G. Variables do not have to be declared before using unless "`use strict;`" pragma (compiler directive) is placed at the beginning of the file. Then all variables must be declared with `my` operator before being used. Good for speeding up program execution and catching typing mistakes ($freed instead of $fred).

Example:
```
use strict;
my $a = "hello"
$b = "goodbye"   # causes compiling error
```

H. Variables are typeless, no distinction between float and string.
```
1.  $a = 1;  $b = "hello";
2.  $c = $a . $a;      # concatenation yields "11"
3.  $c = $a + $a       # addition yields 2
4.  $c = $b . $b;      # concatenation yields "hellohello"
5.  $c = $a + $b;      # addition yields 1 ($b not really a number, treats as 0)
```

6. Rules:
   a) If a string value is used as an operand for a numeric operator, Perl automatically coverts the string to its equivalent numeric value, as if it had been entered as a decimal floating-point value.
   b) Trailing non-numerics and leading whitespace are ignored.
   ie. `" 123.45joe"` converts to 123.45
   c) Something that isn’t a number at all converts to 0. i.e. `"Joe"` is 0
   d) undef converts to 0.

e) When a number is used with a string operator, like concatenation, the value is automatically converted to a string. i.e. 123 becomes `"123"`

III. Operators

A. Assignment

1. `=` is the normal assignment operator
2. `+=  -=  *=  /=  ++  --  %=` are all the same as in C++
3. `.=` is special concatenation operator (see string operators below)

B. Numeric:

1. `+ - * /` are the same as in C++
2. `**` is an exponentiation operator i.e. `2**3` is 2^3
3. `%` is the mod operator and truncates floats automatically before executing
   `$a = 8.54 % 3;` gives 2, which is the same as 8 % 3

C. String:

1. . is the concatenation operator:
   ```
   $a = "Harding" . "Univ";      # gives "HardingUniv"
   ```
2. x is the repetition operator:
   ```
   $a = "abc" x 3;               # gives "abcabcabc"
   $a = 12 x 3;                  # gives "121212"
   $a = (3+2) x 3;               # gives "555"
   ```
3. Variable interpolation - when not ambiguous - otherwise concatenate
   ```
   $a = "Harding";   $b = "$a University";    # gives "Harding University"
   ```

D. Relational:

1. C++ relationals for numeric comparisons: `==  !=  <  <=  >  >=`
2. PL/I relationals for string comparisons: `eq  ne  lt  le  gt  ge`
3. Beware! Use the right comparison for your data...
   ```
   $a = 123;    $b = 32;
   ```
   `if ($a > $b)` yields true `if ($a gt $b)` yields false
4. Special operators for regular expression matching: `=~  !~` (more later)

E. Logical:

1. `||  &&  !` – essentially the same as in C++
2. `or  and  not` – equivalent operators with lower precedence. Also `xor`
   `if (not $a && not $b)` is equivalent to `if (not ($a && (not $b)) )`

IV. Basic I/O

A. Input

```
1.  $a = <STDIN>;      # reads the next line, including the \n
2.  chomp($a);         # removes the \n from the line (safe chop)
3.  chop($a);          # removes one char from end of line
```

B. Output

```
1.  print "hello\n";           # sends hello and newline to STDOUT
2.  print("hello\n");          # identical, parentheses are optional on functions
3.  print($a, " hi ", $b);     # same as print($a." hi ".$b); or print("$a hi $b");
4.  print "cost: \$2.50";      # prints "cost: $2.50".  use \ before special chars
```

5. Beware!
   ```
   print (2+3), "hello";          # prints 5, ignores "hello"
   print ((2+3),"hello");         # works ok
   ```

```
6.  printf("%s %0.2f for %3i", "cost", 163.915, 5);    # like C's printf
                                                        # prints "cost 163.91 for   5"
7.  # This is a "here" document
    print <<END_TOKEN;
    Everything here will be displayed just as it is written.
    $a is interpreted but \$a is not.
    END_TOKEN
```

8. write - special function for formatting reports (not covered here).

V. Control Structures

A. All statements require a block, even if the block only contains one statement. This removes the need for the "dangling else" rule.

B. Choice structures

```
1. if ($a) {$b++}     # {} required for single statement, ";" optional on last line

2. if ($a ne "" && $a ne "0") {      # test is the same as in 1. above
     block1;
   }
   else {
     block2;
   }

3. if (cond1) {
     block1;
   } elsif (cond2) {          # note the spelling: "elsif", not "else if"
     block2;
   } else {
     block3;
   }

4. unless (cond) {block}      # reverses the condition, may also have else

5. $b++ if ($a > 0);          # one-line if condition, no else allowed

6. $b++ unless ($a <= 0);     # equivalent one-line unless condition
```

C. Pretest loop structures

```
1. $i = 1;
   while ($i <= 10) {                # same as in C++, except "" and "0" test
     print $i;
     $i++;
   }

2. $i = 1;
   until ($i > 10) {                 # reverses the condition
     print $i;
     $i++;
   }

3. for ($i = 1; $i <= 10; $i++) {    # just like C++ for statement
      print $i;
   }
```

4. `foreach` - more on this after we cover arrays

D. Posttest loop structures

```
1. $i = 1;
   do {                       # just like C++
     print $i;
     $i++;
   } while ($i <= 10);        # () aren't required around condition
```

```
2.  $i = 1;
    do {
      print $i;
      $i++;
    } until ($i > 10);        # reverses the condition, () optional
```

VI. Arrays (Lists)

A. A list is ordered scalar data. Each element is a separate scalar variable with an independent scalar value. 0 is first element.

B. List literals are comma-separated values enclosed in parentheses:

```
1.  @a = (1, 2, 3);                   # three values
2.  @a = ("joe", 2.7);                # two typeless scalar values
3.  @a = ($b, 21, $c);                # variables are reevaluated when used
4.  @a = ();                          # the empty list
5.  @a = (1..5);                      # list constructor - same as (1, 2, 3, 4, 5)
6.  @a = (1..3, 7, 11..13);           # (1, 2, 3, 7, 11, 12, 13)
7.  @a = ($b .. $c);                  # range determined by values of variables
8.  @a = ("abe", "beth", "chuck", "diane");
    @a = qw(abe beth chuck diane);    # same thing. Quote word function breaks on any
                                      # white space
```

C. Lists may have any size from zero to all available RAM.

D. Arrays which have not been initialized have the empty list value

E. List assignment

```
1.  @list1 = (1, 2, 3);               # three values
2.  @list2 = @list1;                  # all three values are copied
3.  @list3 = (4, 5, @list2, 6);       # six values
4.  ($a, @list1) = @list1;            # $a gets 1st element of @list1 and is removed
```

F. List length

```
1.  $a = @list1;                      # scalar gets length of array
2.  print $#list1 + 1;                # gets subscript of last element (length - 1)
```

G. Undefined values - intermediate elements not assigned are undefined

```
@a = (1, 2, 3);
$a[5] = 4;                    # value is (1, 2, 3, undef, undef, 4), length is 6
```

H. Array element access

```
1.  @a = (1, 2, 3);           # three values
    $b = $a[0];               # $b gets the 1.  Note: use $, not @, for accessing 1 slot
    $c = $a[1];               # $c gets the 2
2.  ($a[0],$a[1]) = ($a[1],$a[0]);    # swaps first two elements
3.  $b = $a[-1];                      # negative subscript works from right end, value is 3
```

I. Array looping structure

```
@a = qw(dog cat pig horse);   # same using default var      # same using for loop
foreach $b (@a) {             foreach (@a) {                for ($i = 0, $i < @a; $i++) {
  print "$b\n";                 print "$_\n";                 print "$a[$i]\n";
}                             }                             }
```

J. Slices (Sublists)

```
1.  @a[2,4]                   # is not a two dimensional array, it is ($a[2], $a[4])
2.  @a[2..4]                  # list ($a[2], $a[3], $a[4])
3.  @a[0,1] = @a[1,0];        # another way to swap elements
4.  @a[4,5] = (9,17);         # change only these two elements
```

K. List functions

```
1.  push(@list, $new);        # add to end of list. Same as @list = (@list, $new);
2.  $old = pop(@list);        # removes rightmost value
```

```
3.  unshift(@list, $new);      # insert at beginning of list. Same as @list = ($new, @list);
4.  $old = shift(@list);       # removes leftmost value. Same as ($old, @list) = @list;
5.  @a = reverse(@b);          # @a is the reverse of @b
6.  @a = sort(@b);             # sorts on string (ASCII) values
7.  chomp(@a);                 # chomps each element of @a
```

L. <STDIN> as an array

```
@a = <STDIN>;                  # gets each line (with \n) in an element
                               # until CTRL-D (Unix) or CTRL-Z (DOS/VMS)
```

M. map and grep

Useful for transforming an entire list. Although syntactically similar, grep is different from map in that it is selective in what it transforms.

```
1. @nums = map $_ + 1, (2, 3, 4);    # adds one to each element returning (3, 4, 5)
2. @names = map { lc $_ } qw(Joe Bill Sue);      # returns (joe, bill, sue)
3. @large = grep { $_ > 3 } @nums;               # returns list of all nums > 3
```

VII. Associative Arrays (Hashes)

A. Description

An associative array (also called a hash) is an array of scalar values where the subscripts are not 0, 1, 2, ... but rather other scalar values in no particular order. For example: `$a{23.7} = 45.2;` and `$zip{"Harding"}=72149;` would store two scalar array values, each with their own scalar subscript. The subscripts (called *keys*) are used later to retrieve the stored values. The elements of a hash are not stored in any particular order in memory. (Perl keeps them in a special order which only Perl knows about! It will find them for you, so you don't need to know the order.)

B. Assigning and referencing hash variables

1. Entire hash variables are referenced using the % - more on this later
2. Individual hash variables (scalars) are referenced using $*name*{*key*}

```
$whatever{"abc"}= 123;        # note { }, not [ ]
$whatever{"xyz"} = "harding";
$key = "abc";
print $whatever{$key};        # prints "123"
print $whatever{xyz};         # prints "harding".  "" are optional inside {}
```

C. Converting lists to hashes and hashes to lists

```
1.  @what = %whatever;              # gives ("abc", 123, "xyz", "harding")
                                    # Not necessarily in that order, but always with keys
                                    # and values paired adjacently
2.  %a = @what;                     # converts list of pairs to a hash
3.  %a = ("tb",3,"sb",4,"sr",2);    # a{"tb"}=3; a{"sb"}=4; a{"sr"}=2;
4.  @a{"tb","sb","sr"} = (3,4,2);   # same thing
5.  %a = (
        "tb" => 3,                  # same thing
        "sb" => 4,
        "sr" => 2);
6.  %b = %a;                        # copies hash
7.  %a = reverse %b;                # swap keys and values, values should be unique
```

D. Hash Functions

1. **keys** - returns a list of the keys

```
    %a = qw(923 bill 335 sam 456 joe);
    @ids = keys(%a);                # returns (923, 335, 456), not
                                    # necessarily in that order


    # print sorted list of keys
    foreach $key (sort (keys (%a))) {
      print "at $key we have $a{$key}\n"; }
```

2. **values** - returns a list of the values
```
@nums = values(%a);    # returns (bill, sam, joe)
```

3. **each** - returns a (key, value) pair in sequence until empty list
   - this is more efficient than using keys
   - do not add or delete elements in this loop (resets & confuses)

```
while (($num, $name) = each(%a)) {
  print "The number for $name is $num.\n";}
```

4. **delete** - removes an element from a hash
```
delete($a{456});              # removes 456-joe from the hash
```

VIII. Functions

A. Defining functions (subroutines)

1. **sub** statement

```
sub subname {                  sub say_hi {
   ...                           print "Hi There!\n";
}                              }
```

2. Naming rules are as for scalars, arrays, and hashes; different namespace.
3. Placement may be before or after main program text (after is preferred).
4. No type associated with function name.
5. No argument list given with function definition - more later.
6. No local subroutines. If there are two with the same name then the 1st is overwritten.

B. Return values

1. **return** statement - very similar to C++ - exceptions noted later.
2. Type of data returned is relaxed, as function has no type associated.
3. May return a scalar or a list.

C. Passing and receiving arguments

1. Arguments are passed in a list with parentheses i.e. `$a = max(2,7);`
2. Arguments are received via a special variable name: the `@_` array
3. `$_[0]` has the first argument, `$_[1]` has the second, etc…
4. `$#_` has the subscript of the last argument
5. undef is given to values beyond the end of the `@_` array.
6. All arguments are passed by reference unless `my` is used (later).
7. Examples:

```
# return the larger argument          # swap two arguments
sub max {                              sub swap {
  if ($_[0] > $_[1]) {                    my $temp;
    return $_[0]; }                       $temp = $_[0];
  else {                                  $_[0] = $_[1];
    return $_[1]; }                       $_[1] = $temp
}                                      }
```

8. Functions with variable numbers of arguments. Consider the following calls to the add function (defined below):
```
a)  $a = add(4,5,6);          # returns 15
b)  print add(1,2,3,4,5);     # prints 15
c)  print add(1..5,7);        # prints 22
```

D. Calling functions

```
1.  say_hi();                        # called as a void function in C++
2.  $x = "first"; $y = "last";
    swap($x, $y);                    # $x will be "last" and $y will be "first"
3.  $a = max(2,7);                   # will return 7
4.  print max(4,9) + max(8,2);       # prints 17
```

E. Private variables

1. **my** operator - causes listed variables to be known in this subroutine only. Saves old value of similarly named global variable and restores value after finishing function.

```
sub add {
   my ($sum, $val);           # $sum and $val are local
   $sum = 0;
   foreach $val (@_) {
     $sum += $val; }
   return $sum;
 }
```

Naming arguments (copying value from `@_`) protects them from being changed (passing by value):

```
sub PrintLNF {                # called PrintLNF("Joe", "Isuzu");
  my ($first, $last) = @_;
  print "Your name is $last, $first.\n";
  $last = "Nice name!";
  print $last, "\n";
}
```

2. **local** operator - semi-private variables - like *my*, but known in the block where defined and also in functions called from the defining block.

IX. File and Directory Access

A. Opening and closing filehandles

1. A filehandle is the name for an I/O connection between your Perl process and the I/O device. These names are used without a special prefix character so it is suggested that you use ALL UPPERCASE to avoid possible collisions with present or future reserved words.

2. Standard handles: `STDIN, STDOUT, STDERR` - opened by default

3. **open** function - returns true if successful in opening a file, false if failed.

```
open(FILEHANDLE, "filename");
```

a)	For reading:	`open(OUT, "myfile.dat");`
b)	For writing:	`open(OUT, ">outfile.dat");`
c)	For appending:	`open(OUT, ">>outfile.dat");`

4. **close** function: `close(OUT);`

B. An aside: the **die** function

```
open(LOG, ">>logfile") || die "cannot append to logfile";
```

Note: `a || b;` is the same as `unless (a) {b};` so code below is the same:

```
unless (open(LOG, ">>logfile") {
  print "Cannot append to logfile" }
```

die causes the execution to be terminated after message to STDERR.
warn is similar - sends message, but continues execution

C. Using filehandles for input

Once a file is opened, you can read lines from it just as you would from STDIN.

Example:
```
open(NAMES, "names.dat");
while ($name = <NAMES>) {
  chomp($name);
  print "The next name is $name.\n";
}
```

Or just use the following to read the entire file:

```
@names = <NAMES>;
```

D. Using file handles for output

```
open(LOGFILE, ">>log.out");
print LOGFILE "finished processing $name";

print STDOUT "hello";  # same as: print "hello";
```

E. File tests

Useful for getting attributes of a file, directory, symlink, etc.

```
$fn = "temp/abc.txt";
print "The file $fn exists." if (-e $fn);
```

File Test	**Meaning**	**File Test**	**Meaning**
`-r -w -x -o`	File or directory is readable, writable, executable, or owned by user	`-b`	Entry is a block-special file (like a mountable disk)
`-R -W -X -O`	File or directory is readable, writable, executable, or owned by real user, not effective user	`-c`	Entry is a character-special file (like an I/O device)
`-e`	File or directory exists	`-u -g`	File or directory is setuid or setgid
`-z`	File exists and has zero size (directories are never empty)	`-k`	File or directory has the sticky bit set
`-s`	File or directory exists and has nonzero size (in bytes)	`-t`	isatty() on the filehandle is true
`-f`	Entry is a plain file	`-T`	File is text
`-d`	Entry is a directory	`-B`	File is binary
`-l`	Entry is a symlink	`-M`	Modification age in days
`-S`	Entry is a socket	`-C`	Inode modification age in days
`-p`	Entry is a named pipe	`-A`	Access age in days

F. Globbing

Definition: the expansion of filename argument patterns into a list of matching filenames.

```
1.  @files = </home/temp/*.c>;              # grabs list of files ending in .c
2.  @files = glob("/home/temp/*.c");        # same thing using glob operator
3.  while ($file_name = <[a-c]*.html>) {    # output each .html file starting with a-c
      print "File: $file_name\n" }
4.  foreach $file_name (<[a-c]*.html>) {    # same thing
      print "File: $file_name\n" }
```

G. File and directory manipulation

```
1.  unlink("myfile.dat");            # removes myfile.dat
2.  unlink <*.o>;                    # same as “rm *.o” in Unix.
3.  rename("oldboy", "newtrick");    # same as “mv oldboy newtrick” in Unix.
                                     # returns false if couldn’t rename
4.  mkdir("datafiles", 0777) ||      # create datafiles with world r/w/e permissions
        warn "Cannot make datafiles directory: $!";
5.  rmdir("datafiles");              # deletes datafiles directory if it’s empty
6.  chmod(0666, "datafiles");        # gives world r/w permissions to datafiles directory
```

X. Regular Expressions

A. A *regular expression* (regex) is a pattern or template which is used to match like strings. Useful for parsing strings and pattern matching.

B. Single-character patterns

```
1.  $name = "Harding University";
    if ($name =~ /ding/) {    # search for "ding" inside of $name
      print "$name\n";}       # and print "Harding University"
    else {
      print "no match\n";}
2.  if ($name !~ /ding/) {    # !~ is the opposite of =~
        print "no match\n”; }
3.  $_ = "Card is hard";      # set default var
    if (/[Hh]ard/)            # search for h or H followed by "ard" in $_
        print;                # print with no arguments prints $_
4.  /d.is/            # match "d" followed by any char (except \n) and then "is"
5.  /\d/              # match any single digit. Same as /[012345789]/ and /[0-9]/
6.  /\w/              # match any single letter or digit. Same as /[a-zA-Z0-9]/
7.  /\s/              # match any single space, tab, newline, carriage return, or form
                      # feed. Same as /[ \t\n\r\f]/
8.  $_ = "agent 07 hello";    # search for space char followed by 2 digits followed by a
    print if (/\s\d\d\s\w+/); # space char followed by one or more word chars

9.  /[^aeiou]/        # ^ negates inside [].  match any single non-vowel
10. /^foo/            # match beginning of string with foo
11. /bar$/            # match end of string with bar
```

C. Grouping patterns

```
1.  /ba*d/            # match zero or more a's between b and d.  Matches "bd" or "bad"
                       # but not "bid"
2.  /ba+d/            # match one or more a's
3.  /ba?d/            # match zero or one a
4.  /b|a|d/           # alternation | means to match exactly one b, a, or d
5.  /(hard|soft)ly/   # match hardly or softly
```

D. Extracting matches

```
1.  "03:15 pm" =~ /(\d\d):(\d\d) (am|pm)/;   # use () to remember what matches
    $hour = $1;                              # $1 matches first () which is "03"
    $min = $2;                                # $2 matches second () which is "15"
    $designation = $3;                        # $3 matches third () which is "pm"
2.  ($hour, $min, $designation) =             # Same thing
       ("03:15 am" =~ /(\d\d):(\d\d) (am|pm)/);
```

E. Substitutions

Format: `s/old-regex/new-string/option`

```
1.  $a = "bisons soon gone?";
    $a =~ s/on/art/;           # replace 1st "on" with "art". $a is now "bisarts soon gone?"
2.  $a =~ s/on/art/g;          # global replace.  $a is now "bisarts soart garte?"
3.  $_ = "Hello, world!";
    $new = "Goodbye";
    s/hello/$new/i;            # case-insensitive match, replace Hello with Goodbye
4.  $_ = "how now cow";
    s/(\w+)/<$1>/g;            # () remember found regexp.  $_ is now "<how> <now> <cow>"
5.  $_ = "who   was there";           # find 1 or more non-space chars at the beginning of
    s/^([^ ]+) +([^ ]+)/$2 $1/;       # string followed by 1 or more spaces followed by 1 or
                                      # more non-space chars and swap.
                                      # Result is "was who there"
6.  s/(^[+-]?)0*(\d+)$/$1$2/;         # converts "0001" to "1", "+00123" to "+123",
                                      # "-2" to "-2"
7.  $_ = "cost +\$1.99";
    s/\+\$\d\.\d\d/free/;             # place \ before reserved chars
    print;                            # prints "cost free"
```

F. **split** function

Returns a list of values that *don't* match a given regex in a search string.

```
1.  $line = "linux:dos::windows:beos";
    @os = split(/:/, $line);          # split $line with : as delimeter
                                      # @os is ("linux","dos","","windows","beos")
    @os = split(/:+/, $line);         # gets rid of empty entry
2.  $_ = "this is a test";
    @words = split(/ /);              # same as split(/ /, $_);
```