An instantaneous introduction to Perl

Michael Grobe

Academic Computing Services

The University of Kansas

July 1999

Topics

Basic Perl statements and variables

Here is a very simple program that uses the Perl print statement:

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

     print "Hello world!";
```

The first line tells the operating system where to find the Perl interpreter. The second line causes the string "Hello world!" to appear on your screen. The second line is composed of a command, `print` and an "argument", `"Hello world!"`, and is terminated by a semicolon (`;`).

Use the pico editor to define a file named `hello.pl` containing this file. Type pico followed by the file name at the UNIX command line prompt, which is probably a dollar sign ($).

```
pico hello.pl
```

After you have entered the program and saved it to the file `hello.pl`, run it by using a command like:

```
     perl hello.pl
```

you should see "Hello world!" appear on your screen, followed by the next command prompt, as in:

```
falcon:/homef/imajhawk$ perl hello.pl
Hello world!falcon:/homef/imajhawk$
```

Now change the print string to "Hello world!\n", and rerun the program. You will then see something like:

```
falcon:/homef/imajhawk$ perl hello.pl
Hello world!
falcon:/homef/imajhawk$
```

This time the program prints a "newline" (represented by "\n") after printing "Hello world!", so that the next command prompt appears on its own line. If you use two "\n" sequences you will create a blank line between the line containing "Hello world!" and the command prompt. Each newline character forces the writing cursor to drop down one line.

Another way to print the same string is to use a print statement like:

```
print "Hello " . "world\n";
```

This statement uses the "concatenation operator" (.) to concatenate the two strings to form a single string containing the same characters in the original.

Yet another way to print this string is to assign it to a Perl "variable", say `$GREETING`, and then print that variable as in:

```
     $GREETING = "Hello world!\n";
     print $GREETING;
```

The variable `$GREETING` is simply a place to store the string for later use. Variables can actually be included within print strings as in:

```
$GREETING = "Hello world!";
print "What I want to say to you is:
$GREETING\n";
```

which will produce something like:

```
falcon:/homef/imajhawk$ perl hello.pl
What I want to say to you is: Hello world!
falcon:/homef/imajhawk$
```

`$GREETING` is a "scalar" variable, a variable holding only one value at a time.

Scalar variable names always begin with a dollar sign ($).

Note also that Perl variables are not "typed". Variable type is determined by use or context.

Reading data from the keyboard

You can read data from the keyboard into a Perl variable by using a program like:

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

    $line = <STDIN>;
    print STDOUT $line;
```

This program will read one line of input from the "standard input" and write that line to the "standard output" which happens to be the screen. Both "STDIN" and "STDOUT" are optional in this context.

You can prompt the user for input by using a print statement followed immediately by a read, as in:

```
#!/usr/local/bin/perl
#
# This program gets a user's age.
#
print "Enter your age:";
$line = <>;  # read a line from the
             # keyboard.

chop $line;  # get rid of the Return.

print "You are $line years old.\n";

exit;
```

When this program runs it will write the prompt string to the screen and wait for you to type in your age and press the `Return` key.

When the input line is placed into the $line variable it ends with a newline representing the `Return` key press.

The Perl `chop()` function was used to eliminate the newline at the end of the reply.

Note the use of the # sign to include comments in the program. Any text to the right of a # sign will be ignored by the Perl interpreter.

Note also the use of the Perl `exit` statement to leave the program. `exit` is optional in this program because there were no more statements to execute, but it is necessary in some programs.

If statements and relational operators

You can evaluate the data input from the user (data validation) by using an "if" statement and a comparison as in:

```
#!/usr/local/bin/perl
print "Enter your age:";

$line = <>;
chop $line;      # get rid of the Return
                 # character.
print "You say you are $line years old.\n";

if ( $line > 130 )
{
    print "I don't believe you!\n";
};

exit;
```

The if statement allows you to compare two numbers and respond accordingly. In this case the number stored in the variable `$line` is compared with the number 130 to see whether `$line` is greater than 130.

If the user enters a number that is greater than 130, the message expressing skepticism will be printed.

The whole expression, `$line > 130` is a "conditional expression", and the general syntax (form) for an if statement may be represented as:

```
if ( conditional_expression )
{
   Perl statements to execute if the
   conditional_expression evaluates "true".
};
```

Another way to handle this validation is to use an if...else... statement like:

```
     #!/usr/local/bin/perl
     print "Enter your age:";

     $line = <>;
     chop $line; # get rid of the Return.

     if ( $line > 130 )
     {
         print "I don't believe you!\n";
     }
     else
     {
  print "You say you are $line years old.\n";
     };
     exit;
```

This time EITHER the message of skepticism OR the message of acceptance will be printed, but NOT both.

You can compare two string values for equality by using the "eq" relational operator, and you can compare two string values for inequality by using the "ne" relational operator, as in:

```
#!/usr/local/bin/perl
print "Enter your name: ";

$line = <>;
chop $line;

if ( $line eq "Megen" )
{
    print "Hi Megen!\n";
}
else
{
    print "Hi dude or dudette.\n";
};
exit;
```

FYI: relational operators return 1 for true and "" for false.

Using logical operators

A user may also enter a negative number, which would be another invalid entry. You can test against both of these invalid entry possibilities by using a logical OR operator (||), as in:

```
     #!/usr/local/bin/perl
     print "Enter your age:";

     $line = <>;
     chop $line;

     if ( $line < 0 || $line > 130 )
     {
         print "I don't believe you!\n";
     }
     else
     {
  print "You say you are $line years old.\n";
     };
     exit;
```

You may also use "or" as the logical OR operator, "&&" or "and" as the logical AND operator, and "!" or "not" as the logical NOT.

For example, you might ask:

```
     if ( $line > 0 && $line < 130 )
```

to identify a believable response.

Actually, to be safe you might use parentheses around every comparison as in:

```
    if ( ( $line > 0 ) && ( $line < 130 ) )
```

to make sure the expression gets evaluated the way you expect it to.

In fact, you MUST use parens in the Megen example, as in:

```
    if ( ! ( $line ne "Megen" ) )
```

The line

```
    if ( ! $line ne "Megen" )
```

will not work because "`! $line`" will be evaluated as one operation, the result of which will be compared with "Megen".

In general, you can define arbitrarily complex conditional expressions by using the comparison and logical operators with appropriate (matched) sets of parentheses.

Arrays

Perl provides special data structures, called "lists" or "arrays", to keep track of lists of things. An array named `@fruits` can be defined as a list of strings by using a Perl statement like:

```
 @fruits = ("apples", "pears", "bananas" );
```

You can then print the value of any element of the list by specifying its location within the list. That is,

```
     print $fruits[2] ;
```

will print "bananas".

You might have expected `$fruits[2]` to get you "pears", but item numbering in Perl lists begins with 0.

Note also that array names begin with "@", but that single elements within an array (i.e., `$fruit[2]`) are usually scalars, so references to them begin with "$".

It is also possible to assign the values of one array to another, as in:

```
 ($red_fruit, $green_fruit, $yellow_fruit) =
           ("apples", "pears", "bananas" );
```

In this case the value of `$red_fruit` will become "apples", `$green_fruit` will become "pears", etc.

Repetition and "looping constructs"

It would be impractical if not impossible to print out the values of large arrays using single print statements. Perl provides special constructs for performing repetitive tasks such as this.

For example, to print the values of each list element within `$fruits` you can write a "foreach loop" like this:

```
#!/usr/local/bin/perl
@fruits = ("apples", "pears", "bananas" );
foreach $fruit ( @fruits )
{
     print "$fruit\n";
};  #end foreach
```

This foreach statement repeats the following process for each element in the array `@fruits`:

- An element of `@fruits` is assigned to `$fruit`,
- The statement(s) surrounded by the curly-braces "{}" are executed.

With this approach you need not know exactly how many elements are contained in a list.

An alternative approach requires foreknowledge of the number of elements and employs a "for loop":

```
#!/usr/local/bin/perl
@fruits = ("apples","pears","bananas" );

for ( $location = 0; $location < 3;
          $location = $location + 1 )
{
     print $fruits[$location] ;
};  #end for
```

This approach implements the following process:

- Set the value of `$location` to 0,
- Continue to print the value of the array element `$fruits[$location]` and increment the value of `$location` by one, as long as $location remains less than 3.

You can modify the second version using the knowledge that the variable `$#fruits` holds the location of the last list element. You would construct the for statement as:

```
for (  $location = 0;
       $location <= $#fruits;
       $location = $location + 1 )
```

You might also use a "while loop" to accomplish the same thing as in:

```
#!/usr/local/bin/perl
@fruits = ("apples", "pears", "bananas" );

$location = 0;
while ( $location < 3 )
{
     print $fruits[$location];
     $location = $location + 1;
};  #end while
```

While loops are especially useful when a program cannot know at the outset how many repetitions it must make. This would be the case when reading information from a file (as you will see later) or from a network connection, etc.

Note that the general form of the while loop provides an `unless` conditional and a `continue` block.

Associative arrays

Perl provides another data structure, called a "hash" or "associative array", to keep track of lists of things.

A hash is an array that indexes each element with a string rather than a number. For example, an associative array named `%fruit_colors` can be defined as:

```
 %fruit_colors = ();  #clear the hash first.

 $fruit_colors{'apple'} = "red";
 $fruit_colors{'pear'} = "green";
 $fruit_colors{'banana'} = "yellow";
```

The name of an associative array begins with a percent sign (%), but a single element with the array is referenced with a name beginning with a dollar sign ($). You can then print the contents of this associative array with a foreach loop like:

```
foreach $fruit (keys %fruit_colors)
{
   print "color of $fruit is
               $fruit_colors{$fruit}\n";
};  #end foreach
```

Here the Perl function `keys()` examines the hash and finds every index value. Within the loop, the color of each fruit is printed directly from the array.

You can also assign values to a hash by using a list of pairs of strings by using a Perl statement like:

```
%fruits = ('apple', "red",
           'pear', "green",
           'banana', "yellow" );
```

or even

```
%fruits = ('apple' => "red",
           'pear' => "green",
           'banana' => "yellow" );
```

Subroutines and modular programming

This program has two "subroutines" that may be called from a "main" program, or "mainline".

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

# This program has two subroutines.

print "Enter your age:";

$line = <>;
chop $line;

if ( ( $line < 0 ) || ( $line > 130 ) )
{
    &print_disbelief;
}
else
{
     &print_age ( $line );
};  #end if
exit;

sub print_disbelief  # sub to show doubt.
{
    print "I don't believe you!\n";
};  #end sub

sub print_age     # sub to print a fact.
{
   my $age = $_[0] ;   # copy argument
   print "You say you are $age years old.\n";
};  #end sub
```

Note that the first subroutine needs no information from the mainline to do its work.

The second one, however, is called with an "argument," the age value entered by the user and stored in `$line`. The subroutine "call"

```
      &print_age( $line );
```

places a pointer to `$line` in the first element of a special array called `@_`, and the information in `$line` can be copied to or modified from the subroutine as `$_[0]`, just as you would access the first element of any "non-special" array.

However, rather than use `$line` itself, `print_age` stores the value of `$line` in a variable called `$age` that is NOT available to any other subroutine OR to the mainline. `$age` is "isolated" from the rest of the program through the use of the "my" statement.

Modular programs are typically easier to understand, debug, and modify because their structure is usually clear and data isolation keeps subroutines from changing each other's data.

Note that the subroutines in the example above are used in contexts that do not require the subroutine to posses or return a value. A subroutine used to produce a value that is used within a calling statement is called a "function". For example, the subroutine `credible()` returns the value "yes" or the value "no".

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

# This program uses a function called
credible().

print "Enter your age:";

$line = <>;
chop $line;

if ( credible( $line ) eq "yes" )
{
    print "Your age is $line.\n";
}
else
{
    print "I don't believe you!\n";
};  #end if
exit;
```

```
sub credible # subroutine to check
credibility.
{
    my $age = $_[0] ;   # copy first argument

    if ( ( $age < 0 ) || ( $age > 130 ) )
    {
        return "no";
    }
    else
    {
        return "yes";
    };  #end if

};  #end sub
```

The function receives `$line` as an argument, compares it with bounding values, 0 and 130, and returns the string "no" or the string "yes". The last value evaluated within the function is the value that will be returned by the function. A Perl function can return a scalar or an array.

Miscellaneous built-in Perl functions

There are many built-in functions available within Perl. See the Perl manual or tutorials for descriptions.

The open and close functions

The `open` function allow your program to "open" or establish a connection to a file (or process) so that information may be exchanged. That is, the program may copy information from the file to program variables, or vice versa.

For example, suppose you have file, called "fruit-colors.txt", containing the following lines:

```
apple red
pear green
banana yello
```

You can "read" that file using a program like:

```
open FRUIT_INFO, "< fruit-colors.txt";

while ( $line = <FRUIT_INFO> )
{
    print $line;
};  #end while

close FRUIT_INFO;
exit;
```

The "<" signifies the program intends to read from the file. That is, the program intends to move data FROM the file TO the program. The following program can be used to create fruit-colors.txt:

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

%fruit_colors = (); #clear the hash first.
$fruit_colors{'apple'} = "red";
$fruit_colors{'pear'} = "green";
$fruit_colors{'banana'} = "yellow";

open FRUIT_INFO, " > fruit-colors.txt";
foreach $fruit (keys %fruit_colors)
{
    print  FRUIT_INFO
       "$fruit $fruit_colors {$fruit}\n";
};  #end foreach

close FRUIT_INFO;
```

The next program opens a connection, called a "pipe", to a process running the `cat` command. It then reads the results back to the script for processing:

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

open FRUIT_INFO,"/usr/bin/cat fruit-colors.txt |";

while ( $line = <FRUIT_INFO> )
{
     print $line;
};  #end while

close FRUIT_INFO;
exit;
```

The system function

The next program uses the Perl `system()` function to use the UNIX shell command "cat" to print the file:

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

system( "/usr/bin/cat fruit-colors.txt" );
```

which will yield:

```
    apple red
    pear green
    banana yellow
```

The same result may be accomplished by a slight variation of the previous example, namely:

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

  print `/usr/bin/cat fruit-colors.txt`;
```

The backtick (`) in the print command identifies the command to be sent to the UNIX shell for execution.

A similar and quite useful alternative is to use:

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

  $data = `/usr/bin/cat fruit-colors.txt`;

  print $data;
```

which has the advantage that the program may manipulate the results of the shell command.

The split function

The next example uses the `split()` function to separate the each input line into parts so they can be used to assign values to elements of a hash:

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

open FRUIT_INFO, ("< fruit-colors.txt" );

while ( $line = <FRUIT_INFO> )
{
    ( $fruit, $color ) = split (" ", $line );
    $fruit_colors{ $fruit } = $color;
};  #end while

foreach $fruit (keys %fruit_colors)
{
     print
    "color of $fruit is $fruit_colors{$fruit}\n";
};  #end foreach

close FRUIT_INFO;
exit;
```

The first part of each line is stored in `$fruit`, and the second part in `$color`. Those two variables are then used to make an entry into the hash `%fruit_colors`. The result of running this program will be:

```
     color of pear is green
     color of banana is yellow
     color of apple is red
```

The join function

The `join()` function reverses the effect of the `split()` function. It will construct a string using separate elements and separate each element with a specified string, as in:

```
#!/usr/local/bin/perl
%fruit_colors = ();    #clear the hash.
$fruit_colors{'apple'} = "red";
$fruit_colors{'pear'} = "green";
$fruit_colors{'banana'} = "yellow";

open FRUIT_INFO, "> fruit-colors.txt";

foreach $fruit (keys %fruit_colors)
{
     $line = join ( " ", ( $fruit,
                  $fruit_colors{$fruit} ) );
     print FRUIT_INFO "$line\n";
};  #end foreach

close FRUIT_INFO;
```

Pattern matches and regular expressions

Consider the example presented earlier which attempted to recognize the name of a user named "Megen". The program compared the information entered by a user with the string "Megen". Of course, the user might enter her name as "Megen Smith", in which case the comparison with "Megen" fails even though the user's name really is Megen.

One way around this is to search for the string "Megen" within the string submitted instead of testing for equality.

You can use a Perl "pattern match" to do just that:

```
#!/usr/local/bin/perl
print "Enter your name: ";

$line = <>;
chop $line;

if ( $line =~ m/Megen/ )
{
    print "Hi Megen!\n";
}
else
{
    print "Hi dude or dudette.\n";
};
exit;
```

Here the binding operator =~ binds a "pattern match" to a specific variable, `$line` in this case). In this context if the user enters a string that contains the substring "Megen" anywhere within it, she will be greeted as "Megen". The pattern match will return either a "true" or "false", just as does a conditional expression used within an if statement. If the pattern match is successful the binding operation will return a "true" logical value.

However, you really can't be sure whether Megen will type her name with a leading capital "M". If she types "megen" or "MEGEN SMITH", the if statement above will not respond with "Hi Megen!" To make sure case sensitivity doesn't confuse things, you could simply add the letter "i" after the search string, making it: `m/Megen/i`. Note also that the "m" is optional in this context.

It is also possible to modify the values in a variable using binding operators. The statement

```
      $line =~ s/Megen/megen/;
```

will "substitute" the string "megen" for the first (leftmost) occurrence of the string "Megen" within the variable `$line`.

```
      $line =~ s/Megen/megen/g;
```

will substitute "megen" for every occurrence of "Megen".

The patterns used within "m" and "s" pattern matches may include characters that have special meanings within the context. For example:

period (.) represents any character; it is a single-character wildcard,

asterisk (*) represents repetition of a character, and

plus (+) represents one or more occurrences of the letter preceeding character or defined collection of characters.

These pattern match strings are known as "regular expressions".

Programming with Perl objects

An "object" is a special kind of data structure that combines data and program components (e.g., subroutines and functions) that use that data. The "object model" provides a way to structure programs to make them more readable, usable, reliable, and re-usable.

Most programming models distinguish between data and the programs using that data, but either do not provide data isolation, implement isolation using complex rules, or inhibit flexibility with their isolation policies.

Most programming languages also provide constructs for modular programming. The object model further encourages or even dictates modular program design.

The Perl features presented to this point are adequate for writing powerful, efficient Perl programs. These features can all be used within the overarching object model.

A "class" is a collection of variable definitions and subroutines that define any object that is a member of that class. Data values contained within an object are called "properties" and subroutines defined within an object are called "methods".

Here is a definition of a class of objects called "person" that would normally be stored in a file called `person.pm`:

```
package person;

sub new
{
     my ($class, $color, $weight ) = @_;

     my $self = {};
     $self = {'Favorite Color'} = $color;
     $self = {'Weight'} = $weight;

     bless $self;
     # bind this object to the person class.

     $self;
};

sub whats_your_favorite_color
{
     my $self = $_[0];
     print "Your favorite color is
         $self->{ 'Favorite Color' }.\n";
};

sub whats_your_weight
{
    my $self = $_[0];
    $self->{ 'Weight' };
};

1;
```

The method "new" is a special method that can be used to "construct" a new instantiation of the class "person". Three arguments are passed to new through the `@_` array (as is common with subroutine invocations), and new uses the line

```
    my ($class, $color, $weight ) = @_;
```

to assign each argument to a corresponding scalar variable reserved for use only by this execution of the subroutine `new()`.

The statement

```
    my $self = {};
```

defines an empty, anonymous hash with `$self`, as a hard reference to the hash. new then assigns values to the two object properties within the new object. For example, the statement

```
   $self = {'Favorite Color'} = $color;
```

assigns the color argument to the "Favorite Color" hash element, and stores the same hard reference into `$self`.

The `new()` function then declares `$self` to be a pointer to an object of the specified class by using the Perl `bless()` function. This is a critical step, since an object in Perl is nothing more than "a blessed thingie." Finally, new returns `$self` to the calling routine.

You can create a new "instance" of a person with a Perl statement like:

```
     $jeff = new person ( 'blue', 160 );
```

or

```
     $jeff = person->new ( 'blue', 160 );
```

Both of these statements will define an object with a favorite color of "blue" and a weight of "160" and store a pointer to that object in the variable `$jeff`.

You can then print information from the object using statements like:

```
     $jeff->whats_your_favorite_color;
```

or

```
     whats_your_favorite_color $jeff;
```

The next example shows a simple program to tell users their favorite color. It uses the class definition provided above in person.pm.

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

use person;
# get the class definitions from the
# person.pm file.

$jeff = person->new( 'blue', 160 );
$michael = person->new( 'red', 170 );
$cole = person->new( 'tangerine', 150 );
```

```
print "Enter your name: ";

while (( $name = <> ) and
                 ( $name ne "quit\n" ) )
{
   chop $name;

   if ( $name =~ m/jeff/i )
   {
      $jeff->whats_your_favorite_color;

      $weight = $jeff->whats_your_weight;
      print "You weigh $weight pounds.\n";
   }
   elsif ( $name =~ m/michael/i )
   {
      $michael->whats_your_favorite_color ;
      $weight = $michael->whats_your_weight;
      print "You weigh $weight pounds.\n";
   }
   elsif ( $name =~ m/cole/i )
   {
      $cole->whats_your_favorite_color ;
      $weight = $cole->whats_your_weight;
      print "You weigh $weight pounds.\n";
   }
   else
   {
      print "I don't know you\n";
   };  #end if

   print "Enter your name: ";
}  #end while

exit;
```

The next version of the same program stores the object locations in a hash, named `%list` for easy access.

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

use person;

$list{'jeff'} = person->new( 'blue', 160);
$list{'michael'} = person->new( 'red', 170);
$list{'cole'} = person->new('tangerine',150);

print "Enter your name: ";

while ( ( $name = <>) and
        ( $name ne "quit\n" ) )
{
   chop $name;

   $found_name = $list{ $name };
   #  see if there is an object
   if ( $found_name ne "" )
   #  associated with that name.
   {
     $list{$name}->whats_your_favorite_color;
     $weight=$list{$name}->whats_your_weight;
     print "You weigh $weight pounds.\n";
   }
   else
   {
     print "I don't know you.\n"
   };  #end if

   print "Enter your name: ";
}  #end while

exit;
```

Some final thoughts

Perl includes many more "features" and functions. Some consider it a veritable kitchen sink of a language in that features seem to appear willy-nilly, with no integrating rationale. Others consider it a godsend and frequently discover features that meet unique needs in special circumstances, and thereby demonstrate a "utilitarian" rationale.

Because of this conglomeration of features, Perl programs can be very hard to comprehend, which makes them difficult to modify or maintain, and sometimes even difficult to write.

Restricting your programs to a consistent structure and a subset of the available Perl statements should help keep your programs tractable.

Michael Grobe
July 1, 1999