More Perl

CS174

Chris Pollett

Oct 25, 2006.

Outline

- Loops
- Arrays
- Hashes
- Functions

Selection Redux

- Last day we learned about how if-else works in Perl.
- Perl does not have a switch statement

Loops

- Like Javascript, Java, and C, Perl has while and for loops:

```
while(condition)
{
   #do something
}
```

```
for(initial expr; control expr; incr expr)
{
   #do something
}
```

- The implicit variable $_ is often used in conjunction with looping. It is the default operator in a function call with no arguments. Consider:

```
while(<STDIN>) #default target of <> is $_
{
   print; chomp; if( $_ eq “Orange”) #print and chomp are applied to $_
    {
       print “Juice\n”;
    }
}
```

Arrays

- A *list* in Perl is an ordered sequence of scalar values.
- A *list literal* is a parenthesized list of scalar values: (3.14, “circle”, 17)
- An *array* in Perl is a variable used to store a list: @arr = (“good”, “bad”, “ugly”);
- Assigning one array to a second array, @a=@b; actually creates a new array object and copies the members.
- When an array is assign to a scalar, it’s length is returned. Ex: $len = @arr;
- One can do list assignments in Perl:

($a, $b, $c, @d) = (“hi”, “there”, “you”, “too”, “are”, “here”);

- Dereferencing arrays is similar to other languages: @b= (“a”, “b”, “c”) $a = $b[1]; #here @b is an array, $a gets value “b”.

More on Arrays

- Suppose we did the commands:

  @list = (2, 4, 6);

  $list[27] =8;

- The list would now have four elements but length 28 (elts 3 through 26 would be vacant).
- The last subscript of a list can be found as $#list. So the length of a list is $#list+1.

Yet More on Arrays

- We can cycle over the elements in a list using foreach:

```
@list = (1, 2, 3);
foreach $value (@list)
{
        print "Value: $value";
}
```

- Perl also has several useful functions for manipulating arrays: shift, unshift (for the front of an array), push, pop (back of array), split, sort.
- For example,

```
$str = "larry curly moe";
@arr = split(" ", $str); # @arr = ("larry", "curly", "moe")
```

- The function qw can be used to quickly create an array of strings:

```
@arr = qw(larry curly moe); #does same as above
```

An example

```
$index = 0;
while($name = <>)
      #above we are getting arguments from the command line
{
   $names[$index++] = uc($name); #upper case $name
}
print “\nThe sorted list of names is:\n\n\n”;
foreach $name (sort @names)
{
    print “$name \n”;
}
```

Hashes

- Associative arrays are arrays in which each data element is paired with a key, which is used to find the element.
- In Perl, associative arrays are called *hashes*.
- hashes differ from arrays in that:
  - one looks up elements in a hash using a string
  - the elements in a hash are not stored in order of subscript
- Here is an example of making a hash:

  %kids_ages = (“John” => 17, “Sammy” => 12, “Sally” =>4);
- To get someone’s age out of this hash we could do:

  $john_age = $kids_ages{“John”};
- To remove an entry from a hash one can do:

  delete $kids_ages{“Sammy”};
- To delete the hash one can use

  %kids_ages =(); #or

  undef %kids_ages;

More on Hashes

- To check if a values is in a hash one can use the exists operator:

```
if(exists $kids_ages{“bob”}){… }
```

- To print out a hash one could do things like:

```
foreach $child (keys %kids_ages)
{
        print “Child $child has age $kids_ages{$child}\n”;
}
```

- To get an array of values we could do things like:

```
@ages = values %kids_ages;
print “The ages are @ages”;
```

- One a hash is embedded in double quotes its keys are not interpolates into the string, to get the keys as well you need to first assign the hash to a list then embed the list.
- Perl has a predefined variable %ENV that store the OS’s environment variables.

References

- A *reference* is a scalar value that reference another variable or literal. So it the value of a reference is an address (i.e., its like a pointer in C, but safer). As an example:

```
$age = 42;
$ref_age = \$age;
@arr = (1, 2, 3);
$ref_arr = \@arr;
```

- To get a reference to an array literal one puts the list in square brackets:

```
$ref_sal = [50000, 29999, 10000];
```

- To get a reference to a hash literal one does:

```
$ref_ages ={‘Curly’ =>31, “More” => 29};
```

- To dereference a reference one can add another $, @, or %.
- So $$ref_age is 42; whereas, $ref_age is an address; similarly, @$ref_arr is the array that is referenced by $ref_arr.
- For arrays and hashes one can do things like:

```
$ref_arr->[0];
$ref_ages->{‘Curly’};
```

Functions

- A Perl *function definition* consists of a function header and a block of code that decribes its action.
- A function header consists of the keyword sub together with the name of the function.
- The block is then a sequence of statements in {}.
- As an example, consider:

```
sub product1
{
    return $first*$second;
}
```

- The above function returns a value explicitly.
- If no return call is explicitly made then the value returned by a function is the value of the last expression evaluated.
- So the next function is equivalent:

```
sub product2 { $first*$second;}
```

- To call a function we could do:

```
$p = product1();
$p = product2();
```

Local Variables

- Variables which are implicitly declared have global scope.
- One can also explicitly define variables in which case their score is local to the enclosing block:

```
my $var; #local to block only --
                # not things called from block
my $count = 0;
my ($num, $sum) = (0,0); #etc
```

- Perl also supports a different kind of local variable

```
local $a =5; #local to block and things called from block
```

More On Functions

- Notice we don't give a parameter list to a Perl function.
- It is passed implicitly through the variable @_.
- For example, consider the code snippet:

```
sub square
{
   my ($arg) = @_;
   return $arg*$arg;
}
print square(4);
```