CS 105 Perl: Hash Manipulation, Lengths, Iteration, and Scoping

Nathan Clement

January 29, 2014
Agenda

Today we will cover several advanced iteration constructs, variable scoping, and a few miscellaneous things.

- **sort**
- Hash manipulation
- Quoting
- Lengths
- @ARGV
- Variable scoping
  - *my*
  - *our*
- Advanced iteration
  - ..
  - grep
  - map
**sort**

`sort` sorts an array or list.

By default, it sorts the array stringwise (like the `eq`, `lt`, etc. operators).

```bash
@b = sort @a;

foreach $foo (sort keys %foo) {
   # ...
}
```
By default, sort works like this:

@bar = sort @foo;
Numeric *sort*

By default, sort works like this:

```plaintext
@bar = sort @foo;
```

# same as

```plaintext
@bar = sort { $a cmp $b } @foo;
```
**Numeric sort**

By default, sort works like this:

```plaintext
@bar = sort @foo;
```

# same as

```plaintext
@bar = sort { $a cmp $b } @foo;
```

cmp is a stringwise comparison operator.

# sort numerically

```plaintext
@bar = sort { $a <=> $b } @foo;
```
Numeric sort

By default, sort works like this:
@bar = sort @foo;

# same as
@bar = sort { $a cmp $b } @foo;

cmp is a stringwise comparison operator.
# sort numerically
@bar = sort { $a <=> $b } @foo;

“spaceship” operator
Writing custom `sort` functions

The code inside the curly braces is a block that will be evaluated to provide the relative position of the two elements. See `perldoc -f sort` for all the details.

The parameters to sort’s comparison block are always `$a` and `$b`. If `$a` and `$b` are already being used, they are automatically hidden inside the comparison block (so that they don’t collide).
delete

Use delete to remove key/value pairs from hashes.

$gone = delete $hash{'}foo'};
@gone = delete @hash{qw(bar baz)};

delete returns the values (not keys) it removes from the hash.
It is legal for a value stored in a hash to be `undef`. `%h = (a => undef);`

For this reason, you should not use `defined` to test for the existence of a hash value for a key. Use `exists` instead.

```perl
defined($h{'a'});  # false
exists ($h{'a'});  # true
```
defined versus exists

\[ \%h = (a \Rightarrow \text{undef}); \]

- `exists($h{'a'})` tests for key-value pair
- `defined($h{'a'})` tests this value
Homework hint

`delete` will only delete a hash-key pair if that key `exists`.  
#  WRONG!
`delete $hash{$key} if exists $hash{$key};`
#  Correct
`delete $hash{$key};`

If that key isn’t in the hash, nothing happens; so don’t check!
Quoting review

You should already know about these quoting operators.

\$a = "\$a \$b\n"; # double quotes interpolate
\$b = '\$a \$b\n'; # single quotes don't
@c = qw(a b);    # this doesn't either
Quoting with arbitrary delimiters

Perl allows interpolative and non-interpolative quoting with arbitrary delimiters. This is convenient if the strings you want to quote have quotes in them.

```
$a = qq($a $b); # double quote
$b = q($a $b);  # single quote
```
Terminating delimiters

If you use a character that has a natural mate, Perl will expect that other character to terminate the string. For example, if you use (, [, {, or <, Perl will expect a ), ], }, or >, respectively. For any other delimiter, Perl will expect that delimiter again.

```
$a = qq<$a $b>; # < and > work together
$b = qq/$a $b/; # / has no natural companion
```
**qw** is a quoting operator, too

Everything we just said in the previous slide also applies to **qw**.

```
@a = qw[foo bar];
@b = qw!baz quux!;
```
Multiplicative concatenation

Sometimes you want a string of a specific length. This is an ideal use case for the $x$ operator!

# I want a string with 40 spaces in it.
$a = '' \times 40; $ # That was easy!
x works with lists and arrays, too

@a = qw(a b) x 3;
@b = (@a, 'c') x 3;
Length: strings

To get the length of a string, use `length`.

```
length('' x 40); # returns 40
```
Length: arrays

To get the length of an array, use `scalar`.
An array evaluated in scalar context results in the number of elements in the array.

```
@a = qw(a) x 25;
scalar(@a); # returns 25
```

The `scalar` function forces scalar context.
Context example: array in loop test

Remember that 0 is false.

while(@a) {
    # remove an element of @a
}

Once @a is empty, the loop will terminate, since an empty array will evaluate to 0 in a scalar context, which is false.
Use `scalar keys %hash` to get the number of keys in a hash. You can use this to test if a hash is empty, just as with arrays.

```perl
# WRONG
while(%a) {
    # delete from %a ?
}

# Correct
while(scalar keys %a) {
    # delete from %a
}
```

You can omit the `scalar` here, since `while` forces a scalar context.
I want the Truth!

Now that we know about how scalar context works for plurals (lists, arrays, etc.), we can cross one of the five false values off our list.

**Four** values in Perl are false.

- `undef`
- ""
- 0
- "0"
- "0"

Everything else is true.

In Lecture 2, ( ) was also on this list, but in a boolean context (a subtype of scalar context), it will evaluate as 0.
Scoping

In Perl, unless you make an effort, all variables are global.

This means that the variable can be seen throughout the whole program once it is defined.
my

Introduce a local variable with my.

open my $fd, '<', 'filename' or die;
my @a = qw(apple bear car dog egg fan goat);
my $c;

foreach my $a (@a) {
    my $line = <>;
}

When in doubt, use my.
A variable is global by default. However, if you see a variable assignment in a program, it may not be clear whether it is a local or a global.

our solves this problem. It declares a global variable. This way you know that it’s global on purpose.

our @a = qw(apple bear car dog egg fan goat);
Using blocks for scoping

Sometimes you need a variable for a very short period of time.

Put a pair of curly brackets around its lifetime!

```perl
{ my $foo = $baz * 10;
  my $bar = sprintf('-- %70s --', $foo);
  $hash{$foo} = $bar;
}
```
Danger: multiple declaration

If you declare a variable with `my` twice, you’re actually creating a whole new variable both times.

# WRONG:

```perl
{ my $foo = $baz x 10;
  # ...
  my $foo;
  # ...
  $hash{$foo} = $bar; # oops.
}
```
The command line arguments to a program are automatically assigned to `@ARGV`. 

```perl
foreach $arg (@ARGV) {
    # ...
}

$first = shift;

shift without an argument defaults to shifting from `@ARGV`.

@ARGV
reverse

reverse reverses a list.

@a = qw(5 4 3 2 1);
@a = reverse 1..5; # same thing
.. the range operator

The .. operator fills in elements of a list between two numbers.

@a = qw(1 2 3 4 5);
a = 1..5; # same thing
.. the range operator

The .. operator fills in elements of a list between two numbers.

@a = qw(1 2 3 4 5);
@a = 1..5; # same thing

This is also helpful for loops:

for (1..10) {

the range operator

The .. operator fills in elements of a list between two numbers.

@a = qw(1 2 3 4 5);
@a = 1..5; # same thing

This is also helpful for loops:

for (1..10) {
    say "Do this ten times.";
.. the range operator

The .. operator fills in elements of a list between two numbers.

@a = qw(1 2 3 4 5);
@a = 1..5; # same thing

This is also helpful for loops:

```perl
for (1..10) {
    say "Do this ten times.";
}
```
.. the range operator

• Why didn’t this work?
  for (@items[1..-1]) {
    print "$_\n";
  }

the range operator

• Why didn’t this work?
for (@items[1..-1]) {
  print "$_\n";
}

• How do we solve it?
the range operator

• Why didn’t this work?
for (@items[1..-1]) {
    print "$_\n";
}

• How do we solve it?
for (@items[1.@items-2]) {
    print "$_\n";
}
**grep filters a list**

Use **grep** to select only the elements of an array or list that evaluate to true given some evaluation code.

### Before
@b = ();
foreach $a (@a) {
    next if boring($a);
    push @b, $a;
}

### After
@b = grep { !boring($_) } @a;
map runs some code for each array element

map can be used as a succinct foreach replacement.

# Before
foreach $a (@a) {
    next if boring($a);
    say $a;
}

map runs some code for each array element

map can be used as a succinct **foreach** replacement.

# Before
foreach $a (@a) {
    next if boring($a);
    say $a;
}

# After
map { say } grep { !boring($_) } @a;
map runs some code for each array element

map can be used as a succinct foreach replacement.

# Before
foreach $a (@a) {
    next if boring($a);
    say $a;
}

# After
map { say } grep { !boring($_) } @a;

# After a bit of golfing
map { boring($_) || say } @a;
map: the return values pass through

map is more useful when applying a computation.

# Before
foreach $a (1..10) {
    push @b, $a * $a;
}
# After
@b = map { $_ * $_ } 1..10;

The values returned from each iteration are collected in a single list.
map example

map can return an arbitrary number of values.

One possibility: hash key and value.

@a = qw(apple bear car dog egg fan goat);
# Before
foreach $a (@a) {
    $b{$a} = ++$c;
}
# After
%b = map {$_ => ++$c} @a;

The code block returns both the key and the value.
When to use?

- **foreach**: modify the variable in-place, iterate read-only over each element of a list
- **grep**: filter or count a list
  - **want a resulting subset or count**
- **map**: transform one list to another
  - **want a resulting list**
- From PerlMonks: *It should not transform the items on the list as a matter of practice; it should not be used with side effects nor should it be used as the way to iterate through a list.*
Which one?

```perl
my @a;
for (@b) {
    push @a, $_[0] if condition($_);
}
```
Which one?

my @a;
for (@b) {
  push @a, $_ if condition($_);
}

Which one?

my @a;
for (@b) {
    push @a, $_ if condition($_);
}

my @a = grep {condition($_)} @b;
Which one?

my @a;
for (@b) {
    push @a, $_ x 4;
}

Which one?

my @a;
for (@b) {
    push @a, $_ x 4;
}

my @a = map {$_ x 4} @b;
Homework 3

• Posted
• Due in one week
• 80% correctness, 20% style