CS 105 Perl: 
File I/O, slices, and array manipulation

Nathan Clement

January 27, 2013
Agenda

• Intermediate iteration
  – `last` and `next`
• Intermediate I/O
• Special variables
• Array manipulation
  – `push`, `pop`, `shift`, `unshift`
  – Slices
  – Flattening
  – `splice`
Homeworks

• Homework 1
  • “I thought about printing out War and Peace, but decided not to”

• Homework 2
  • Precipitation units
  • 70% correctness, 30% style
  • Posting on Piazza
last

last immediately exits a loop, like **break** in Java.

```perl
foreach $baz (@baz) {
    if ($baz == $special) {
        last;
    }
    # ...
}

Better:
```
next ends the current iteration and starts the next, like continue in Java.

```plaintext
foreach $baz (@baz) {
  if (boring($baz)) {
    next;
  }
  # ...
}
```

Better:

```plaintext
foreach $baz (@baz) {
  next if boring($baz);
  # ...
}
```
The default variable: \$_

If you don’t specify a variable name for a `foreach` loop, Perl will automatically use \$_.

```perl
foreach (@baz) {
    next if boring($$_);
    # ...
}
```

Many syntactical constructs and built-in functions in Perl use \$_ by default.
$_ example: chomp

chomp chomps $_ by default.
What you used to do:
while(defined($line = <STDIN>)) {
   chomp $line;
   # ...
}

Now:
while(defined($_ = <STDIN>)) {
   chomp;
   # ...
}
< > inside a `while`

Perl will use a `defined`-ness test in a `while` loop when the source is a `< >` I/O operator.

Before:
```perl
while(defined($_ = <STDIN>)) {
  chomp;
  # ...
}
```

After:
```perl
while($_ = <STDIN>) {
  chomp;
  # ...
}
```

Identical behavior, Perl being dwimmy
$_ and the < > I/O operator

Perl will automatically assign $_ to the result of <>, as long as it’s just inside the while conditional.

```perl
while($_ = <STDIN>) {
    chomp;
    # ...
}
```

```perl
while(<STDIN>) { # Better
    chomp;
    # ...
}
```
<ARGV>, or just <>

<> is Perl’s ultimate dwimmy I/O operator. Short for <ARGV>, if will automatically read from the arguments on the command line, or STDIN if none are present.

while(<ARGV>) {
  chomp;
  # ...
}

An idiomatic Perl I/O loop:
while(<>){  # Huzzah!
  chomp;
  # ...
}
To read or write a specific file, use `open`.

```perl
open(CONFIG, 'config');
while(<CONFIG>) {
    chomp;
    # ...
}
close(CONFIG);
```

Don't forget to close files when you're done with them, too.

*This is the “old way”: the 2-argument `open` and bareword file handles.*
open, somewhat improved

Since Perl 5.6 (2000), file handles can be stored in scalars.

```perl
open($config, 'config');
while(<$config>) {
    chomp;
    # ...
}
close($config);
```
open, almost fully modernized

Again since Perl 5.6, open has a 3-argument version where the second argument indicates the ‘mode’ (read, write, pipe, etc.). This is the preferred way of opening files in Perl.

```perl
open $config, '<', 'config';
while(<<$config>) {
    chomp;
    # ...
}
close($config);
```

The only thing missing now is a scoping directive on the scalar.
`split` is a built-in function that breaks a string into separate strings (scalars). By default, it splits `$_` wherever there is whitespace.

```perl
while(<>) {
    chomp;
    @words = split;
    # ...
}
close($config);

For example:
$_ = 'high low up down left right';
@words = split;
# @words = ('high', 'low', 'up', 'down',
#           'left', 'right');
```
Special variables

Perl has many variables that are special or magical; \$_ is just one of them.

They are all documented in perlvar.
$0

$0 is automatically set to the name of the program. For instance, if my program is named `foo` and I run it with `. / foo`, then $0 = './foo'.

This is very handy when combined with a built-in function called `die`.

die "usage: $0 <input> <output>\n" unless ...
More uses for **die**

Sometimes you might get some input that you just can’t use, but you still can’t ignore it.

```perl
while(<>) {
    chomp;
    die "..." if epic_fail($_);
}
```

Oftentimes failure to open a file is fatal:

```perl
open $config, '<', 'config'
    or die "Could not open config: $!";
```
**push**

**push** adds elements to the end of an array.

```plaintext
push @a, 10;  # push 10 onto the end of @a
push @a, @b;  # push the entire contents of @b
```
pop

`pop` removes the last element of an array and returns it.

```perl
$last = pop @a;
```
**shift and unshift**

**shift** and **unshift** are like **push** and **pop**, respectively, but add and remove elements from the *beginning* of the array.

```plaintext
unshift @b, @c;
unshift @b, 1;  # insert 1 at $b[0]

$first = shift @b;  # $first = 1
```
Handy array function table

<table>
<thead>
<tr>
<th>Array operation</th>
<th>at front</th>
<th>at end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add element(s)</td>
<td>unshift</td>
<td>push</td>
</tr>
<tr>
<td>Remove element</td>
<td>shift</td>
<td>pop</td>
</tr>
</tbody>
</table>
Perl allows you to access multiple elements from an array, hash, or list. This subset of scalars in a container is called a slice.

```perl
($first, $last) = @a[0, -1];    # accesses @a
($al, $bob) = @a{'al', 'bob'}; # accesses %a
```

Remember the **sigil rule**!
Slices as lvalues

We can assign to slices as well.

@days[0,6] = ('Sunday','Saturday');
@d ogs{ 'p oodle' , 'lab rador' } = ( 'f ierce' );
List flattening

When putting arrays in a list, the elements are expanded into the containing list. The array does not maintain its identity.

Another way of saying this is that you can't create a hierarchy of arrays (arrays inside arrays) this way.

@a = (@b, @c, $a, keys %d);
Perl provides an easy way to create a list using the `qw` operator.

```perl
@days[0,6] = ('Sunday', 'Saturday');
@dogs{'poodle','labrador'} = ('fierce');

Can be written as
@days[0,6] = qw(Sunday Saturday);
@dogs{qw(poodle labrador)} = qw(fierce);
```
**splice**

**splice** is a powerful function that allows you to remove scalars at any location in an array and/or insert new scalars at that point.

splice ARRAY, OFFSET, LENGTH, LIST

<table>
<thead>
<tr>
<th>splice(@a, -1)</th>
<th>pop(@a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>splice(@a, 0, 1)</td>
<td>shift(@a)</td>
</tr>
<tr>
<td>splice(@a, 0, 0, $x, $y)</td>
<td>unshift(@a, $x, $y)</td>
</tr>
<tr>
<td>splice(@a, $i, 1, $y)</td>
<td>$a[$i] = $y</td>
</tr>
<tr>
<td>splice(@a, @a, 0, $x, $y)</td>
<td>push(@a, $x, $y)</td>
</tr>
</tbody>
</table>
splice

my ($winner, $runnerup) = splice(@finalists, 0, 2);

my @others = qw(SnowWhite Humbert);
my @dwarfs = qw(Doc Grumpy Happy Sleepy Sneezy Dopey Bashful);
splice @dwarfs, 2, 4, @others;
print "@dwarfs\n";
# Doc Grumpy SnowWhite Humbert Bashful
Practice

<table>
<thead>
<tr>
<th>Name</th>
<th>Sale 1</th>
<th>Sale 2</th>
<th>Commission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark</td>
<td>20030</td>
<td>19958</td>
<td>19.24</td>
</tr>
<tr>
<td>Tony</td>
<td>19636</td>
<td>19038</td>
<td>21000 16.10</td>
</tr>
</tbody>
</table>

Require input data from a file: cars.txt

Delimited by tabs