Programming in Perl

Week Four

References and aggregate data structures

Documentation

Debugging

Exercise 3.1

```
#!/usr/bin/perl -w
use strict;

my %users;
while(<DATA>){
    chomp;
    my ($user, $time) = split /:/;
    $users{$user} += $time;
}

foreach my $user (keys %users){
    print "User: $user\n";
    print "Total time online: $users{$user}\n\n";
}

__DATA__
bjones:27
asmith:102
asmith:12
jdoe:311
bjones:45
```

Exercise 3.2

```
sub sum {
    my $sum = 0;
    foreach my $arg ( @_ ) {
        $sum += $arg;
    }
    return $sum;
}
```

Exercise 3.3

```
sub max_num {
    my $max = shift;
    foreach my $item (@_) {
        $max = $item if $item > $max;
    }
    return $max;
}

sub min_num {
    my $min = shift;
    foreach my $item (@_) {
        $min = $item if $item < $min;
    }
    return $min;
}
```

Exercise 3.4

```
sub max_num {
    my $max = shift;
    foreach my $item (@_) {
        $max = $item if $item gt $max;
    }
    return $max;
}

sub min_num {
    my $min = shift;
    foreach my $item (@_) {
        $min = $item if $item lt $min;
    }
    return $min;
}
```

References

- In Exercise 3.1 we had a data file that looked like this:

```
bjones:27:termA
asmith:102:termB
asmith:12:termA
jdoe:311:termC
bjones:45:termA
```

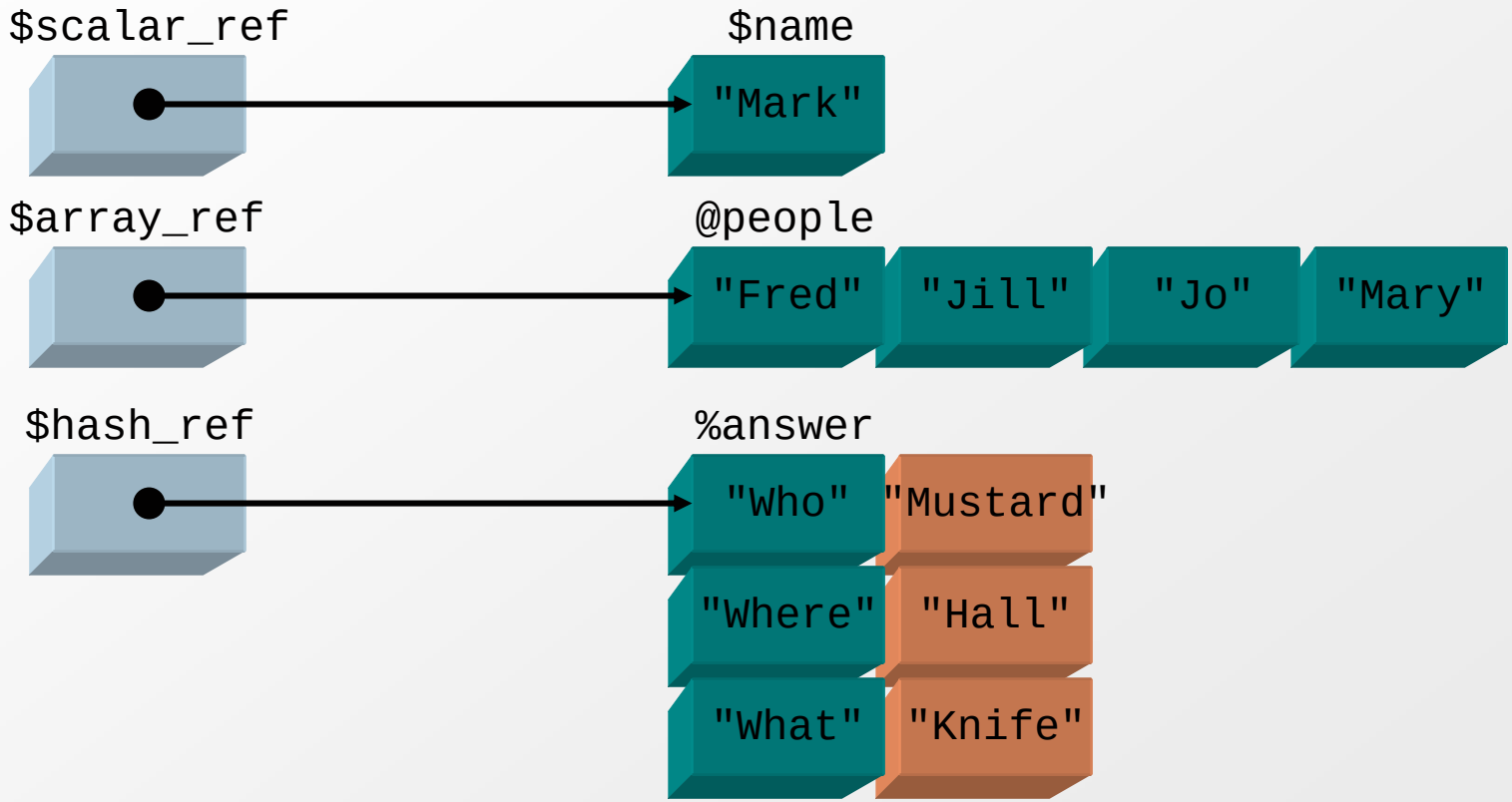
```
($name, $time) = split /:/;
$user{$name} = $time;
```

What Are References

- References are a way to use a variable indirectly
- A reference “points” to the value of variable without knowing the name of the variable
- A reference is always contained in a scalar variable (or a scalar element in an array or hash)

What Are References

- You can point to any type of data that Perl knows about



Named References

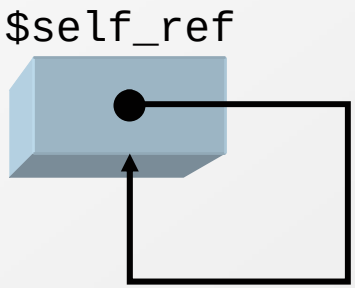
- Named references point to a variable that has been already been declared in the script.
- You can create references to variables by adding a backslash (“\”) before the variable

```
$name = "Mark";
$scalar_ref = \$name;

@people = ("Fred", "Jill", "Jo", "Mary");
$array_ref = \@people;

%hash = (Who => "Mustard", Where => "Hall", What => "Knife");
$hash_ref = \%answer;
```

Named References

- You can even create a reference to itself!

```
$self_ref = \$self_ref;
```



Anonymous Array and Hash References

- Anonymous references allow you to point to non-named arrays and hashes
- You create a reference to an anonymous array or hash by using a special syntax
  - Arrays are surrounded with square brackets ([])

```
$anon_array = [ "Fred", "Jill", "Joe", "Mary" ];
```

  - Hashes are surrounded with curly braces ({})

```
$anon_hash = { Who => "Mustard", Where => "Hall", How => "Knife" };
```

Anonymous Array and Hash References

- Anonymous arrays and hashes will live as long as there is a reference to them. Perl will automatically clean up any un-referenced anonymous array or hash

Accessing Through References

- Accessing data through a reference in a variable is called "de-referencing"
- To de-reference, simply prefix the variable with the proper symbol ($, @, or %)

```
$$array_ref[2] = $$hash_ref{'Who'};
```

- A more general, and less ambiguous, way to write the same thing

```
${$array_ref}[2] = ${$hash_ref}{'Who'};
```

- To get to scalars (and other things) you must use this de-referencing syntax

```
$$scalar_ref = "Fred";
```

The Arrow Operator (->)

- When you are working with references to arrays or hashes, you can use the arrow operator (->)between the variable that holds the reference and the subscript

```
$array_ref->[2] = $hash_ref->{'Who'};
```

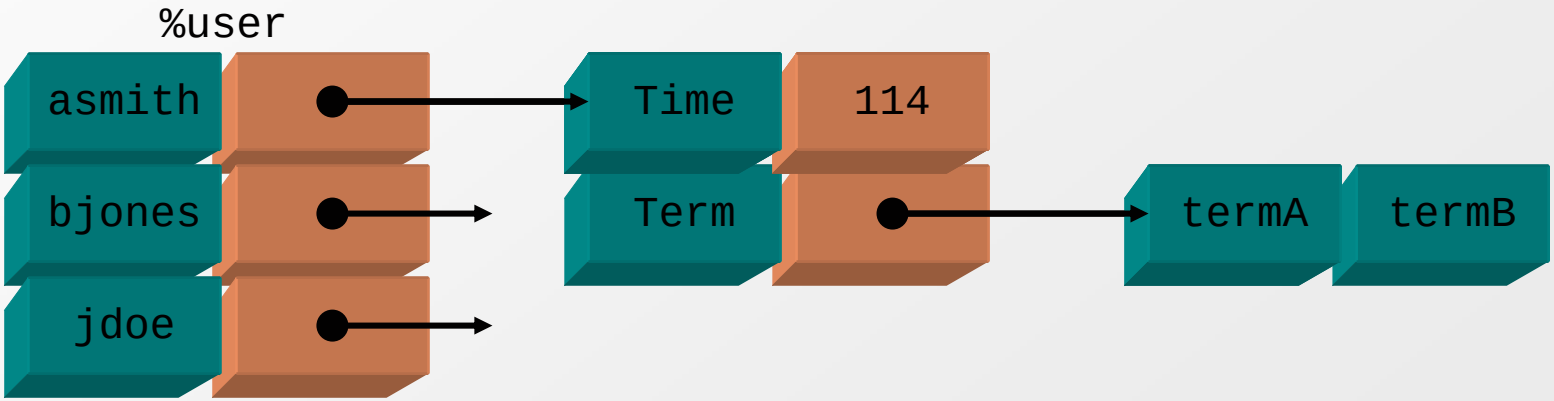
- Remember, the arrow operator *only* works with arrays or hashes

Using References

- We can now write our user tracker!

```
asmith:102:termB
bjones:27:termA
asmith:12:termA
jdoe:311:termC
bjones:45:termA
```

```
($name, $time, $term) = split /:/;
$user{$name}->{Time} += $time;
unshift @{$user{$name}->{Term}}, $term;
```



Building Complex Data Types

- Using arrays or hashes that contain references to other arrays or hashes, gives you a way to build complex data types
- An example: Perl does not have a built-in data type that can handle multi-dimensional arrays, but you can build one using an array that contains references to arrays that contain row information
- This is call a “List of Lists” or LoL

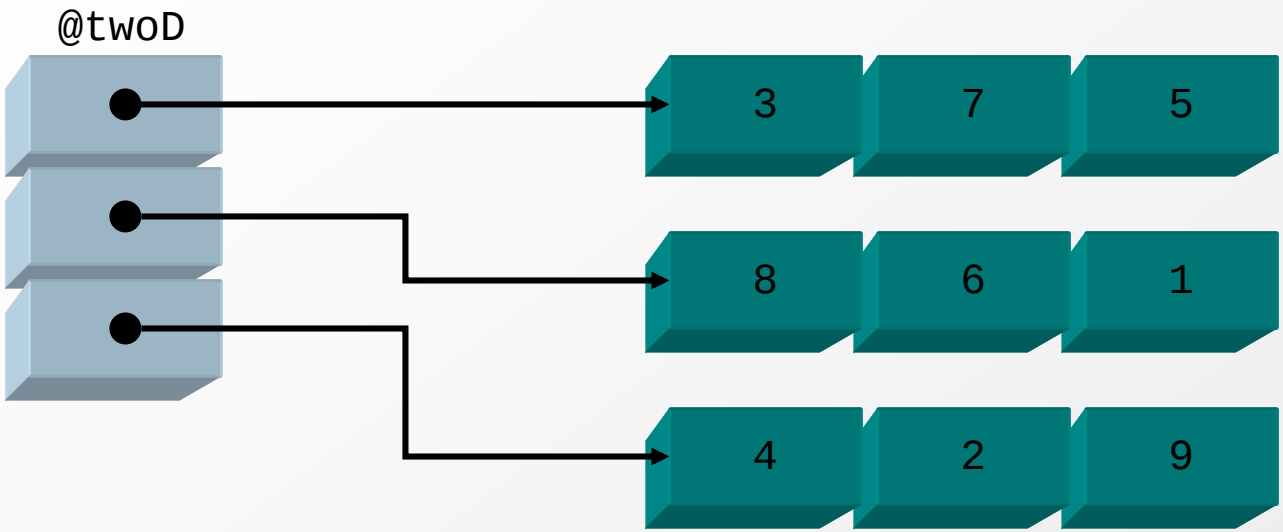
List of Lists

- A two dimensional array can be build like:

```
@twoD = (
            [3, 7, 5],
            [8, 6, 1],
            [4, 2, 9]
        );
```

- Each element in `@twoD` is a reference to an array that represents a row in the two dimensional array, and each element in the row arrays represents a column

List of Lists



- To access

```
print $twoD[0][1];    # Prints 7
print $twoD[2][2];    # Prints 9
```

Identifying a Referent

- You can identify what kind of reference you have by using the `ref()` operator.

Perl Documentation

- Perl has a way to embed user level documentation (User Guides, etc.) into your scripts using a mark up language call Plain Old Documentation, POD for short
- There are several programs that extract this documentation from scripts to make printable or online documents
- `pod2text` and `pod2html` are included with Perl. I wrote `pod2fm` to create FrameMaker documents

POD

- PODs are driven by “paragraphs” where paragraphs are divided by one or more blank lines
- There are three paragraph types

```
=head1 THIS IS A COMMAND PARAGRAPH

This is an ordinary paragraph.
Even though it stretches over several lines, all
of the lines are included in the paragraph.

    This is a verbatim paragraph.
    It will be reproduced from the script
    exactly as it is shown here. Each line must
    be indented!
```

Command

Ordinary

Verbatim

Command Paragraphs

- Command paragraphs start with a ‘=‘ followed by an identifier and the arbitrary text

Identifier	Desicription
`=head1 heading` `=head2 heading`	First an second level headings. The following paragraphs are treated as normal paragraphs in the text.

`=over N`
`=item text`
`=back`
	Indent the next paragraphs with `=over` until you get to a `=back`. `=item *` will produce a bullet in front of the next paragraph, and `=item 1`, `=item 2` will produce a number list. `=item text` will create an "hanging indent" with the text as the hang.

`=pod` `=cut`	You and insert POD into arbitrary spots in your script by putting it within `=pod É =cut`.
`=for X` `=begin X` `=end X`	`=for` allows you to sent the next paragraph directly to the formatter, without the POD interpretation. `=begin É =end` work the same way for multiple paragraphs. `=for html  ` `<p> This is a raw HTM paragraph </p>`

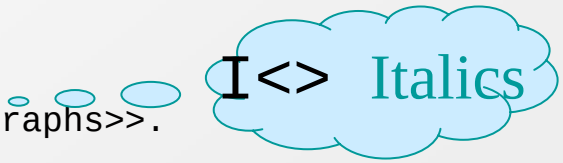
Verbatim Paragraphs

- Verbatim paragraphs are repeated, err…, verbatim
- You just need to indent the lines you want repeated with any formatting.

```
This is an Ordinary paragraph. It will
be formatted!!

    #!/usr/local/bin/perl -w
    use strict;

    while (<>) {
        print;
    }
```

Ordinary Paragraphs

- Ordinary paragraphs will be formatted and maybe justified
- You can include interior sequences that can modify the formatting



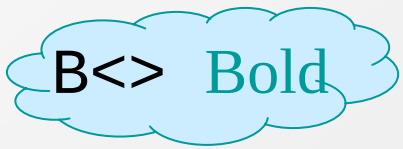

```
This paragraph is a good test of PODs S<I<Ordinary paragraphs>>.
It includes B<Interior Sequences>, and a line code:
C<$a E<lt>=E<gt> $b>
```

This paragraph is a good test of PODs *Ordinary Êpaagraphs*. It includes **Interior Sequences**, and a line code: `$a <=> $b`

Ordinary Paragraphs

- Ordinary paragraphs will be formatted and maybe justified
- You can include interior sequences that can modify the formatting

```
This paragraph is a good test of PODs S<I<Ordinary paragraphs>>.
It includes B<Interior Sequences>, and a line code:
C<$a E<lt>=E<gt> $b>
```

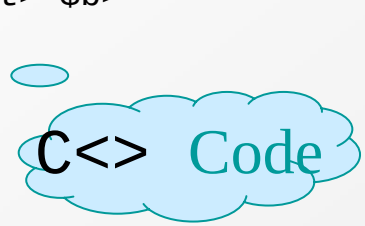


This paragraph is a good test of PODs *OrdinaryÊparagraphs*. It includes **Interior Sequences**, and a line code: $a <=> $b

Ordinary Paragraphs

- Ordinary paragraphs will be formatted and maybe justified
- You can include interior sequences that can modify the formatting

```
This paragraph is a good test of PODs S<I<Ordinary paragraphs>>.
It includes B<Interior Sequences>, and a line code:
C<$a E<lt>=E<gt> $b>
```



This paragraph is a good test of PODs *OrdinaryÊparagraphs*. It includes **Interior Sequences**, and a line code: `$a <=> $b`

Ordinary Paragraphs

- Ordinary paragraphs will be formatted and maybe justified
- You can include interior sequences that can modify the formatting

```
This paragraph is a good test of PODs S<I<Ordinary paragraphs>>.
It includes B<Interior Sequences>, and a line code:
C<$a E<lt>=E<gt> $b>
```

S<> Non Breaking Space

This paragraph is a good test of PODs *OrdinaryÊparagraphs*. It includes **Interior Sequences**, and a line code: $a <=> $b

Ordinary Paragraphs

- Ordinary paragraphs will be formatted and maybe justified
- You can include interior sequences that can modify the formatting

```
This paragraph is a good test of PODs S<I<Ordinary paragraphs>>.
It includes B<Interior Sequences>, and a line code:
C<$a E<lt>=E<gt> $b>
```

E<> Escape

This paragraph is a good test of PODs *Ordinary paragraphs*. It includes **Interior Sequences**, and a line code: $a <=> $b

Using the Perl Debugger

- Perl comes with an interactive, line based, debugger

```
perl -d foo.pl
```

Class Project

- For your class project, you need to write a non-trivial Perl script that:
  - Works with a file, reading it into a complex data structure, and writing it back out
  - Edit the complex data structure
  - Have embedded user documentation
  - Use at least one of Perl supplied modules
  - Should be at least 200 Perl Code lines long

Project Examples

- An Address Book that allows you to enter new addresses and edit old, and prints out mailing labels
- A CD or Book index that records the CD or book with the track name and time (or table of contents/page number) that prints index cards
- A filter that reads in an E-mail message, that may or may not contain a quoted message, and “cleans up” the quoted message

Project Proposal

- For next week, write a Draft Project Proposal in the form a user's manual in POD
- Name the file `<project>.pod`
- It must have:

```
=head1 NAME
=head1 SYNOPSIS
=head1 DESCRIPTION
=head1 OPTIONS
=head1 FILES
=head1 AUTHOR
```

- It should be 1 to 3 pages long

Homework 4.1

- Write a program that will take a two dimensional array, transpose it, and print out the results

One	Two	Three
Four	Five	Six
Seven	Eight	Nine

One	Four	Seven
Two	Five	Eight
Three	Six	Nine