A Crash Course in Perl5

Part 3: Data structures

Zeegee Software Inc.
http://www.zeegee.com/

Terms and Conditions

Road map

- Basics
  - Introduction
  - Perl syntax
  - Basic data types
  - Basic operators
- Patterns
  - Introduction
  - String matching and modifying
  - Pattern variables
- Data structures
  - LISTs and arrays
  - Context
  - Hashes
- Flow control
  - Program structures
  - Subroutines
  - References
  - Error handling
- Data
  - Input and output
  - Binary data
  - Special variables
- Object-oriented programming
  - Modules
  - Objects
  - Inheritance
  - Tying

Data structures

LISTs and arrays

LISTs

- **LISTs** are comma-separated sequences of scalars (enclosed in parentheses when precedence requires)

```
chmod 0755, 'ls', 'dir'; # a list
chmod(0755, 'ls', 'dir');      # same list
```

- Not data types *per se*, but used in...
  - Subroutine/method calls, to pass arguments:

```
print $x, " is greater than ", $y;
```

  - Array/hash initialization:

```
@exts = ('gif', 'tif', 'ps', 'xbm');
```

Arrays

- **Arrays** hold sequences of 0 or more **scalars**
- Indexing is done with **square brackets** (`[]`)
- Indexing starts at **zero** (**0**)
- **Array variables** are signified by a `@` before the variable name: **`@names`**
- **Array elements** are scalars, so put a `$` before the variable name when you refer to them: **`$names[0]`**
- **Array slices** are arrays, so put a `@` before the variable name when you refer to them: **`@names[0..2]`**

The value of a LIST

- When assigning a LIST to a **scalar**, the value of the list literal is the value of the final element:

```
$last = ('Mercury', 'Venus', 'Mars');
print $last, "\n";
```

```
Mars
```

- When assigning a LIST to an **array**, the entire list is assigned to the array:

```
@all = ('Mercury', 'Venus', 'Mars');
print $all[2], "\n";
```

```
Mars
```

Wordlists

- If you want to initialize from a list of strings which are all single-quoted with no whitespace, you can use the special **qw{}** (quote wordlist) operator:

```
# These are identical:
@all =   ('Mercury', 'Venus', 'Mars');
@all = qw(Mercury Venus Mars );
```

- You can use the same kinds of delimiters as with **q{}**, **qq{}**, etc.

Interpolation in LISTs

- When a LIST is evaluated, each element of the LIST is evaluated (in a list context) and the resulting list value is **interpolated** (spliced) into the LIST:

```
@a2c = ('A', 'B', 'C');
$d   =  'D';
@e2z = ('E'..'W', 'X'..'Z');

# This sets @all to the array ('A'..'Z'):
@all = (@a2c, $d, @e2z);
```

- Interpolating an empty list/array has no effect.

Assigning to LISTs

- LISTs may be assigned to only if each element of the list is legal to assign to (i.e., is an lvalue):

```
($a, $b, $c) = (0, 1, 2);
($a, $b, $c) = @somearray;
```

- The final element may be an array or a hash:

```
($a, $b, @rest) = (0, 1, 2, 3, 4, 5);
($a, $b, %rest) = (0, 1, 2, 3, 4, 5);
```

- Elements which are not assigned get the **undef** value:

```
($a, $b, $c) = (0, 1);      # c is now undefined
```

Initializing arrays

- Arrays may be initialized by a LIST of 0 or more elements, enclosed in parentheses:

```
@empty = ();# empty array
@names = (         # array of 6 elements:
      "Kirk", "Bones", "Spock",
      "Uhura", "Rand", "Chapel"
);
```

- Or, from **slices** of other arrays (notice the @s!):

```
@men    = @names[0,1,2];
@women  = @names[3..5];
@medics = @names[1,5];
```

The length of an array

- To get the *number of elements*, evaluate the array in a scalar context. You can do this by assigning to a scalar variable, or by using the function **scalar()**:

```
@names = ("Uhura", "Rand", "Chapel");
$numnames = scalar(@names);   # set to 3
```

- To get the *index of the last element*, use `$#` in front of the array variable name... *without* the @_:

```
@names = ("Uhura", "Rand", "Chapel");
$lasti = $#names;             # set to 2
```

Changing an array's length

- Surprise! The `$#arrayname` variable may be assigned to, to extend or shorten the array:

```
@names = ("Uhura", "Rand", "Chapel");
$all = join('/', @names);  # Uhura/Rand/Chapel
```

```
$#names = 4;
$all = join('/', @names);  # Uhura/Rand/Chapel//
```

```
$#names = 1;
$all = join('/', @names);  # Uhura/Rand
```

- If extending an array, all previously-unassigned slots are set to the undefined value

Fun with arrays

```
@names    = ();                    # not needed, but nice
@suffixes = ('', '-A', '-B', '-C', '-D');
```

```
print scalar(@names), "\n";    # prints 0
for ($i = 0; $i < scalar(@suffixes); $i++) {
      $names[$i] = "NCC1701$i";
}
print scalar(@names), "\n";    # prints 5
```

```
print "$names[4]\n";           # prints NCC1701-D
$names[4] = "Enterprise-$suffixes[4]";
print "$names[4]\n";           # prints Enterprise-D
```

split(*PATTERN, EXPR, LIMIT*)

- Splits a string into an array of strings, via a given pattern:

```
$fullname = 'Captain James T. Kirk';
@parts = split(/\s+/, $fullname);
print "$parts[0] $parts[3]";
```

```
Captain Kirk
```

- Can set a limit on the number of elements to be returned:

```
($rank, $name) = split(/\s+/, $fullname, 2);
print "$rank: <$name>";
```

```
Captain: <James T. Kirk>
```

join(*EXPR, LIST*)

- Joins a list (or array) of strings into a single string:

```
@parts = ('enterprise', 'starfleet', 'ufp');
$hostname = join('.', @parts);
print "picard\@$hostname\n";
```

```
picard@enterprise.starfleet.ufp
```

- Remember, list interpolation is your friend!

```
@parts = ('enterprise', 'starfleet', 'ufp');
$hostname = join('.', 'mail', 'bridge', @parts);
print "picard\@$hostname\n";
```

```
picard@mail.bridge.enterprise.starfleet.ufp
```

push(*ARRAY, LIST*) / pop(*ARRAY*)

- **push()** adds one or more elements to the *right side* (the end) of an array:

```
@digits = (1..7);
push(@digits, 8, 9, 10);    # @digits is now (1..10)
```

- **pop()** removes one element from the *right side* (the end) of an array, and returns it (or `undef` if array is empty):

```
@digits = (1..10);
$last = pop(@digits);          # @digits is now (1..9)
                               # $last is now '10'
```

unshift(*ARRAY, LIST*) / shift(*ARRAY*)

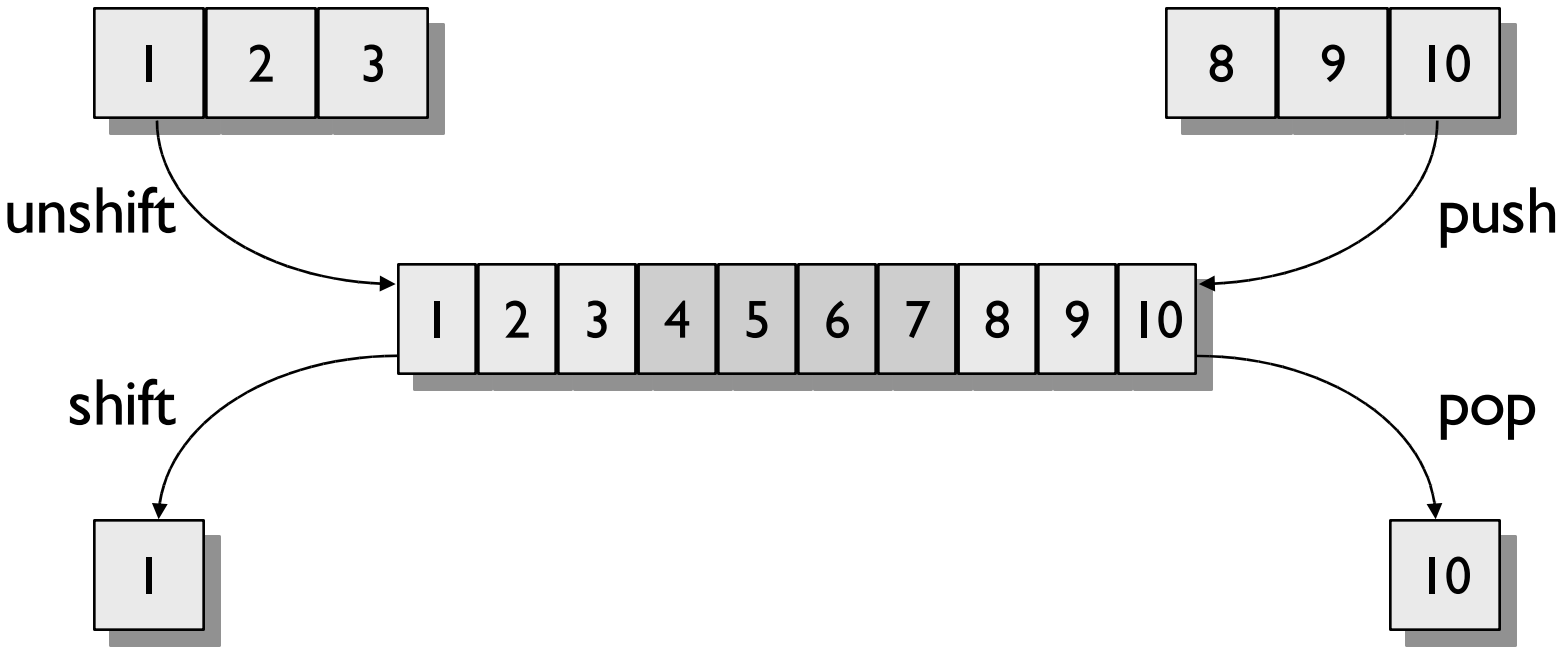
- **unshift()** adds one or more elements to the *left side* (beginning) of an array:

```
@digits = (4..10);
unshift(@digits, 1, 2, 3); # @digits is now (1..10)
```

- **shift()** removes one element from the *left side* (beginning) of an array, and returns it (or `undef` if array is empty):

```
@digits = (1..10);
$first = shift(@digits);    # @digits is now (2..10)
                            # $first is now '1'
```

Shifting vs. push/pop



sort(*LIST*)

- **sort()** takes a LIST and sorts it in ascending string-comparison (**cmp**) order, *returning* the sorted list value:

```
@raw = ('Z', 'B', 'A', 'X', 'C', 'Y');
@sorted = sort @raw;
print join('-', @sorted);    # prints A-B-C-X-Y-Z
```

The original list remains unsorted!

```
print join('-', @raw);       # prints Z-B-A-X-C-Y
```

Changing sort()'s order

- You can provide a *block* to change the sorted order, using special variables `$a` and `$b`:

```
@raw = ('Z', 'B', 'A', 'X', 'C', 'Y');
@ascending  = sort { $a cmp $b } @raw;
@descending = sort { $b cmp $a } @raw;
```

```
@raw = (21, 3, 567, 1, 10, 100, 1000);
@ascending = sort { $a <=> $b } @raw;
```

- You can also supply a *subroutine name*:

```
sub backwards { $b cmp $a }
@descending = sort backwards @raw;
```

reverse(*LIST*)

- In a *list context*, returns a list value consisting of the elements of LIST in reversed order:

```
@fwd = ('A', 'B', 'C');
@rev = reverse @fwd;      # @rev is now 'C', 'B', 'A'
```

- In a *scalar context*, reverses the bytes of the first element of LIST, and returns that:

```
$rev = reverse 'Picard';
print $rev;
```

```
draciP
```

Data structures

Context

List context vs. scalar context

- The interpretation of operations/values sometimes depends on the **context** around the operation/value
- Two major contexts: **scalar** and **list**
  - Some operations return list values in list context, and scalar values in scalar context; *or, in other words...*
  - Perl overloads some operations based on whether the expected return value is **singular** or **plural**
- An operator provides either a scalar or list context to each of its arguments: this affects what gets sent in!

Examples of context

- Context determines what an array's "value" is... for example, when assigning it to something else:

```
$size = @names;      # value in scalar context is length
@copy = @names;      # value in list context is the array
```

- Context determines what the <> input operator does:

```
# Causes a single line to be read from STDIN:
$one = <STDIN>;

# Causes all lines to be read from STDIN!
@all = <STDIN>;
```

Context and arguments

- Assignment op uses left side to set context of right side:

```
$size        = @names;   # assign to scalar = scalar context
@copy        = @names;   # assign to array = list context
@copy[2..4]  = @names;   # assign to array slice = list context
($a, $b)     = @names;   # assign to list = list context
```

A function provides either a scalar or list context to each of its args: read the documentation!

`print LIST;`

Beware! The argument to print is a LIST, so all the elements of the argument list are evaluated in a LIST context:

```
print "All elems: ", @elems, "\n";
```

Context and arguments (cont'd)

Usage: `print $expr, @x, $len;`

Declaration: **`print LIST;`**

Meaning: This function takes a LIST, so each of its arguments is evaluated in a LIST context. This means that the *elements* of `@x` will be interpolated into the argument list between $expr and $len.

Usage: `substr $expr, @x, $len;`

Declaration: **`substr EXPR, OFFSET, LENGTH;`**

Meaning: This function takes three scalars, so each of its arguments is evaluated in a SCALAR context. This means that the *number of elements* of `@x` is used as the second argument.

Context and return values

There is no general rule for converting a list value to a scalar, so read the documentation if you're going to call a list-oriented operator in a scalar context!

```
$x = sort @somearray;        # what is $x?
```

- For functions that can be called in both contexts, failure is usually indicated by returning...

`undef`	...in the **scalar** context
( ) *(the null list)*	...in the **list** context

What's *my* context?

- User subroutines can ask about their context and act accordingly, using **wantarray():**

```
sub alphabits {
  return (wantarray ? ('A','B','C') : 'XYZ');
}
@arr = alphabits();                    # sets @arr to ('A',
'B', 'C')
$str = alphabits();            # sets $str to 'XYZ'
```

- Use it to return an appropriate error value:

```
return (wantarray ? () : undef);
```

- Generally, though, you don't need to worry about it.

Forcing scalar context

- You can force scalar context via the **scalar()** operator:

```
@counts = (scalar @a, scalar @b);
```

- If the scalar you want is an integer anyway, you can also use the **int()** operator... it's considered sloppy and inefficient, but it's nicer to type and it drives the point home:

```
print "Count = ", int(@a), "!\n";
```

Data structures

Hashes

What are hashes?

- **Hashes** (or **associative arrays**) are much like arrays, except that instead of storing/retrieving information by a numeric index (e.g., 0, 1, 2), you store/retrieve information by a text string, called the *key*.
- A quick comparison of arrays and hashes...

```
$array[12] = 'Kirk';
$capn = $array[12];
```

@array is an array

```
$hash{'Captain'} = 'Kirk';
$capn = $hash{'Captain'};
```

%hash is a hash

Hashes

- **Hashes** consist of zero or more **key=value** pairs, where the keys and values are all **scalars.** You store and retrieve a given value by its key.
- Only **one value per key**.
- Indexing is done with **curly brackets** (`{ }`)
- **Hash variables** are signified by a `%` before the variable name: **`%names`**
- **Hash values** are scalars, so put a `$` before the variable name when you refer to them: **`$names{'Captain'}`**

Initializing hashes

- Like arrays, hashes may be initiallized by a LIST of 0 or more elements, enclosed in parentheses:

```
%names = (
      'Captain',          'Kirk',
      'Science Officer', 'Spock',
      'Medical Officer', 'Bones'
);
```

- The elements are regarded as

  (key1, val1, key2, val2, ..., keyN, valN).

- There must be an even number of elements!

The => operator

- Perl provides the => list operator. It is (almost) syntactically identical to the comma (,) but is more readable when initializing hashes:

```
%names = (
      'Captain'         => 'Kirk',
      'Science Officer' => 'Spock',
      'Medical Officer' => 'Bones'
);
```

- The list elements are now more obviously read as...

(key1 => val1, key2 => val2, ..., keyN => valN).

Barewords as hash keys

- The use of `=>` as a hash initializer is known to Perl... you can safely use barewords on the left hand side (if they are not all-lowercase):

```
%names = (
      Captain        => 'Kirk',
      ScienceOfficer => 'Spock',
      MedicalOfficer => 'Bones'
);
```

- Ditto for hash keys inside curly braces:

```
$capn = $names{Captain};
```

keys(*HASH*)

- Takes a hash and returns all the keys, as an array:

```
%crew = ('Captain'     => 'Kirk',
         'Helm'        => 'Sulu',
         'Chief Nurse' => 'Chapel');
@stations = keys %crew;
print join('/', @stations);
```

```
Helm/Chief Nurse/Captain
```

- Keys are returned in an apparently random order, but it is the same order as in **values()** and **each()**
- In a scalar context, **keys()** returns the *number* of keys

More fun with keys()

- Can be used in conjunction with sort():

```
# Print out all environment vars, sorted:
foreach $key (sort keys %ENV) {
      print "$key = $ENV{$key}\n";
}
```

```
HOME = /home/eryq
PATH = /bin:/usr/bin:/usr/local/bin
SHELL = tcsh
USER = eryq
```

values(*HASH*)

- Takes a hash and returns all the values, as an array:

```
%crew = ('Captain'     => 'Kirk',
         'Helm'        => 'Sulu',
         'Chief Nurse' => 'Chapel');
@names = values %crew;
print join('/', @names);
```

```
Sulu/Chapel/Kirk
```

- Values are returned in an apparently random order, but it is the same order as in **keys()** and **each()**
- In a scalar context, **values()** returns the *number* of values

each(*HASH*)

- Takes a hash and returns a 2-element array of the next *key=>value* pair, for iterating over the hash:

```
# Print out all environment variables:
while (($key, $value) = each %ENV) {
      print "$key = $value\n";
}
```

- When no more pairs remain, returns the empty array... which when assigned as shown has a FALSE value.

Don't modify the hash while iterating over it!