COMP 2021

Unix and Script Programming

Perl Arrays and Lists

Perl Array

- Store a list (any number of ordered scalars).
- Have a name preceded by an "@"

```
@fibonacci = (1, 1, 2, 3, 5, 8, 11);           # Numbers
@fruits = ("apples", "bananas", "cherries");   #Strings
@grade_synonyms = (100, "A++", "Perfect");     #Both
```

- Indexed by number
  - First item is index 0, last item is index `$#array`
  - Negative numbers count from end of list

```
@fruits = ("apples", "bananas", "cherries");
print "Fruit flies like $fruits[1].\n";
print "Life is like a bowl of $fruits[$#fruits].\n";
print "We need more $fruits[-3] to make the pie.\n";
$fruits[0] = "oranges"; # Replace apples with oranges
```

Perl Array (cont.)

- Items can be assigned in several ways
  - Items bounded by parentheses and separated by commas
  - Numeric value ranges denoted by .. operator
  - Quoted word lists using `qw` operator
  - Sublists are "flattened" into a single array

```
@prime_numbers = (2, 3, 5, 7, 11, 13);          # Comma-separated
@composite_numbers = (4, 6, 8..10, 12, 14..16); # Numeric ranges

@fruits = ("apples", "bananas", "cherries");
@fruits = qw(apples bananas cherries);          # Same as above

@veggies = qw(radishes spinach);
@grocery_list = (@fruits, @veggies, "milk");
print "@grocery_list\n";
# print out "apples bananas cherries radishes spinach milk"
```

Array Values and Size

- Dynamically assume whatever values or sizes need
  - May be defined, but empty
  - Can dynamically lengthen or shorten arrays
  - `unshift` and `shift` add to and remove from the front
  - `push` and `pop` add to and remove from the end

```
my @fruits;                                  # Undefined
@fruits = qw(apples bananas cherries); # Assigned
@fruits = (@fruits, "dates");           # Lengthen
@fruits = ();                                  # Empty

unshift @fruits, "acorn";                # Add an item to the front
my $nut = shift @fruits;                  # Remove from the front
print "Well, a squirrel would think an $nut was a fruit.\n";

push @fruits, "mango";                    # Add an item to the end
my $food = pop @fruits;                   # Remove from the end
print "My, that was a yummy $food!\n";
```

Scalar and List Context

- If an operator or function expects a scalar argument, the argument is evaluated in a *scalar context*.

```
$n = @nums;      # $n gets the length of @nums
```

- If an operator or function expects a list argument, the argument is evaluated in a *list context*.

```
($n) = @nums; # $n gets the first element of @nums
```

- A scalar value used within a list context is promoted to a single-element array.

```
@nums = 1;  # @nums = (1)
```

Bad Subscripting

- If you access an array beyond the end of the array, the `undef` value is returned without warning.
- `undef` is 0 when used as a number, the empty string when used as a string.

```
@nums = (5,6,7);
$nums[3] = "cindy"; # (5,6,7,"cindy")
$nums[5] = "li";    # (5,6,7,"cindy",undef,"li")
```

- Assignment to an array element with a subscript less than zero is a fatal error.

Array Slices

- Accessing a list of elements from the same array is called a *slice*.
- Perl provides a special shortcut for slices:

```
@a = (1,2,3);
@a[0,1] = @a[1,0];      # swap the first two elements
@a[0,1,2] = @a[1,1,1]; # make all 3 elements like the 2nd
@a[1,2] = (7,4);        # change the last two to 7 and 4
                         # a: (1,7,4)
```

- Note that slices use `@` rather than `$`. This is because slices work with lists rather than scalar values.

Array Slices (cont.)

- Slices also work directly on lists

```
$c = (1,2,3,4,5)[2];          # sets $c to 3
@b = (1,2,3,4,5)[2,4];        # sets @b to (3,5)
```

  - The second statement above is equivalent to:

```
@x = (1,2,3,4,5);
@b = @x[2,4];
```

- You can use also array expressions to index slices if you want to be tricky

```
@nums = (5,6,7);
@i = (2,1,0);
@rev = @nums[@i];             # @rev is now (7, 6, 5)
```

Left-Side Assignment

- If a list only contains variables, you can use it on the left side of an assignment:

```
($a,$b,$c) = (1,2,3);          # set $a=1, $b=2, $c=3
($a,$b) = ($b,$a);             # swap $a and $b
($d,@arr) = ($a,$b,$c);        # set $d=$a and @arr=($b,$c)
($e,@arr) = @arr;              # remove 1st element of @arr and
                               # put it in $e
```

- An array variable can only occur in the last position in the list, because the array variable is "greedy" and consumes all the remaining values.

`reverse` and `sort`

- The `reverse` function reverses the array, returning the resulting list

```
@a = (1,2,3);
@b = reverse(@a);       # @b=(3,2,1), @a unchanged
@b = reverse(1,2,3);    # same thing
```

- The `sort` function returns a sorted array in ascending ASCII order

```
@size = qw(small medium large);
@sortsize = sort(@size); # large, medium, small
@sortsize = sort(qw(small medium large));  # same
@a = (1,2,4,8,16,32,64);
@b = sort(@a);          # @b=(1,16,2,32,4,64,8)
```

Example: Max Value

- Find the biggest value in an array of numbers `@a`

```
@a = (23,4,56,99,36,24);
$max = $a[0];
for($i=1; $i<@a; $i++){
        if($a[$i] > $max){
                $max = $a[$i];
        }
}
print "max is: $max\n";
```

Iteration inside the Array

- `foreach` loop iterates over entire array
- Good to localize the scalar to the loop

```
@names= qw (Foo Bar Baz);
# loop of content
foreach my $name(@names) {
    print “$name\n";
}
# loop index
foreach my $i (0 .. $#names) {
      print “$i - $names[$i]\n”;
}
```

Brain Storming: Take n Elements Randomly from Perl Array

- Task: Given integer array A = [a1, a2, a3, …., aP] with size P. Assume elements are different with each other. Sample q elements from array A.