Arrays and Loops

Sofia Robb

An array is a Named Ordered List.

- What is a list?
  - (‘cat’, ‘dog’, ‘narwhal’)
- Why is it named?
  - @animals = (‘cat’, ‘dog’, ‘narwhal’);
- Why is is ordered?
  - each element has an ordered numerical index or position

Arrays are denoted with ‘@’ symbol

0	1	2
cat	dog	narwhal

Arrays

- Each element of an array has to be a scalar variable
- These are all scalar variables
  - number
  - letter
  - word
  - sentence
  - $scalar_variable

Example array

```
my @colors = ('red', $favorite_color,
'cornflower blue', 5);
```

The elements of the array are stored in a specific order.

```
my @colors = ('red', $favorite_color,
'cornflower blue', 5);
```

0	1	2	3
'red'	$favorite_color	'cornflower blue'	5

$colors[0] $colors[1] $colors[2] $colors[3]

Each element of an array can be accessed by its position, or index, in the array.

```
my @colors = ('red', $favorite_color,
'cornflower blue', 5);
```

GET

```
my $first  = $colors[0];
my $second = $colors[1];
my $third  = $colors[2];
my $last   = $colors[-1];
```

negative numbers can be used to access from the end

The value of each element can be reassigned with use of its index.

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
$colors[0] = 'green';
$colors[2] = 'gray';
```

$colors[0]	$colors[1]	$colors[2]	$colors[3]
green	$favorite_ color	gray	5

Assign values to indices that are far away

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
$colors[0] = 'green';
$colors[2] = 'gray';
$colors[8] = 'black';
```

$colors[0]	$colors[1]	$colors[2]	$colors[3]	$colors[4]	$colors[5]	$colors[6]	$colors[7]	$colors[8]
green	$favorite_ color	gray	5	undefined	undefined	undefined	undefined	black

@colors now contains 9 elements.
4 of the elements are undefined.

GET/SET: Mirror Images

```
#GET:
$first  = $colors[0];
$second = $colors[1];
```

```
#SET:
$colors[0] = 'green';
$colors[2] = 'gray';
```

A common MISTAKE is to try to access an element in array context (meaning using the '@').

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

This is wrong:

```
my $first  = @colors[0];
```

This is correct:

```
my $first  = $colors[0];
```

Length of an array

`scalar(@array)`

The scalar() function can be used to return the scalar attribute of an array. It scalar attribute is the length, or in other words, the number of elements in the array.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);

my $length = scalar @colors;
print "len of array: $length\n";
```

Output:

```
len of array: 4
```

A common MISTAKE is to use the length() function to get the number of elements in an array

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```

WRONG:

```
my $length = length @colors;
print "len of array: $length\n";
```

Output:

```
len of array: 1
```

CORRECT:

```
my $length = scalar @colors;
print "len of array: $length\n";
```

Output:

```
len of array: 4
```

Quick print of an array

When an array is printed with use of double quotes ("@array"), a single white space is automatically inserted between each element. This allows for a quick way to visualize the contents of your array.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);

print "@colors";
```

Output

```
red purple cornflower blue 5
```

Notice that the print out of the array looks like it has 5 elements while our array actually has 4 elements. Printing within quotes may not always be helpful in cases when a white space is included within a single element, such as 'cornflower blue'.

Array to a String

```
my $new_string = join(string , @array);
```

join() can be used to combine all the individual elements of a list or an array into a string on a set of characters. A string is returned.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
my $new_string = join ('--' , @colors);
print "$new_string\n";
```

Output

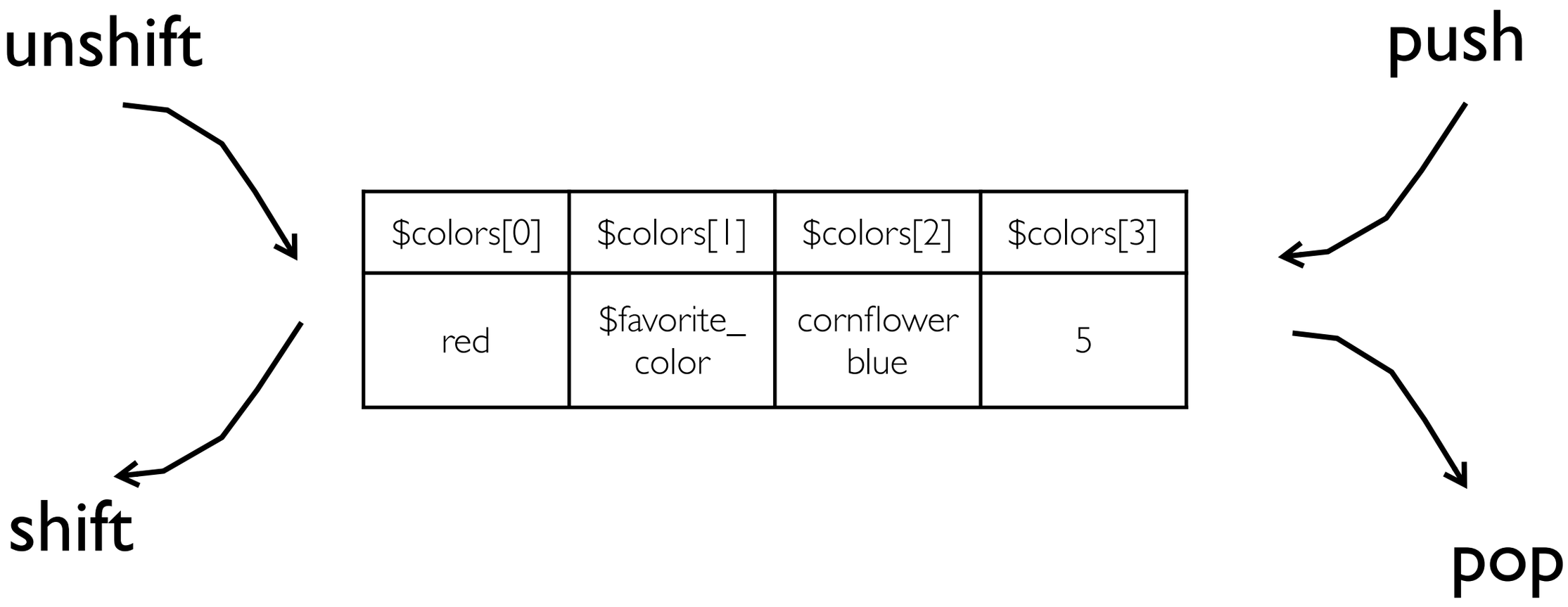
```
red--purple--cornflower blue--5
```

'--' is used here to clearly differentiate the elements of @colors. A tab ("\t") is a common character to use with the join() fuction.

Arrays are Dynamic

Not only can values be reassigned but
Arrays can grow and shrink.

```
my @colors = ('red', $favorite_color, 'cornflower blue', 5);
```



shift() has been used in previous lectures to get user command line arguments

Add elements to the end with push();

```
push (@array, list of values);
```

push

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add one element to the end
push (@colors, 'black');
print join ('--', @colors) , "\n";
```

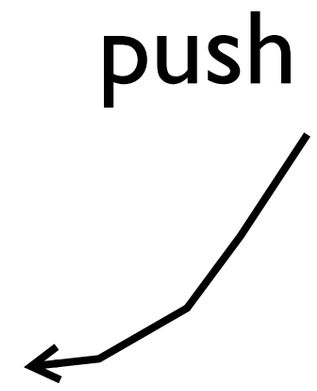
Output

```
red--purple--cornflower blue--5--black
```

push() is changing the actual array

Add elements to the end with push();

```
push (@array, list of values);
```



$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add multiple elements to the end
push (@colors, 'black','blue');
print join ('--', @colors) , "\n";
```

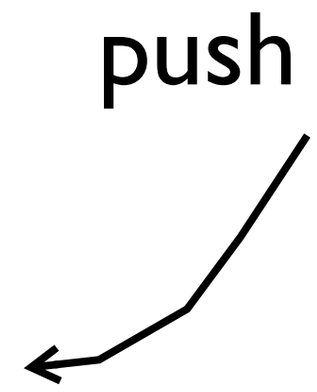
Output

```
red--purple--cornflower blue--5--black--blue
```

push() is changing the actual array

Add elements to the end with push();

```
push (@array, list of values);
```



$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add an array of elements
my @more_colors = ('yellow','pink','white','orange');
push (@colors, @more_colors);
print join ('--', @colors) , "\n";
```

Output

```
red--purple--cornflower blue--5--black--yellow--pink--white--orange
```

push() is changing the actual array

Remove an element from the end with pop();

```
my $last = pop(@array);
```

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

pop

```
my $last_element = pop @colors;

print "last: $last_element\n";
print join ('--', @colors) , "\n";
```

Output

```
last: 5
red--purple--cornflower blue
```

pop() is changing the actual array

Remove an element from the beginning with shift();

```
$first = shift(@array);
```


```
my $first_element = shift(@colors);

print "first: $first_element\n";
print join ('--', @colors) , "\n";
```

Output

```
first: red
purple--cornflower blue--5
```

shift() is changing the actual array

Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add one element to the beginning
unshift (@colors, 'black');
print join ('--', @colors) , "\n";
```

Output

```
black--red--purple--cornflower blue--5
```

Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add two elements to the beginning
unshift (@colors, 'black' , 'blue');
print join('--',@colors), "\n";
```

Output

```
black--blue--red--purple--cornflower blue--5
```

Add elements to the beginning with unshift();

```
unshift (@array, list of values);
```

unshift

$colors[0]	$colors[1]	$colors[2]	$colors[3]
red	$favorite_ color	cornflower blue	5

```
#add an array of elements to the beginning
my @more_colors = ('yellow','pink','white','orange');

unshift (@colors, @more_colors);
print join('--',@colors) , "\n";
```

Output

```
yellow--pink--white--orange--red--purple--cornflower blue--5
```

Dynamic Arrays

Function	Meaning
push(@array, a list of values)	add value(s) to the end of the list
$popped_value = pop(@array)	remove a value from the end of the list
$shifted_value = shift(@array)	remove a value from the front of the list
unshift(@array, a list of values)	add value(s) to the front of the list
splice(...)	everything above and more!

String to an Array

```
my @array = split(/pattern/ , string);
```

The split() function can be used to create an array from a string by providing a delimiter of any set of characters or any pattern. split() is similar to Excel's "Text to columns" feature that allows you to indicate which characters separate each field, such as tabs (\t) and commas (,). Just like in Excel, the split() function will remove the delimiter, and it will not be present in the returned data.

```
my $string = "I do not like green eggs and ham";

#'/ /' sets the delimiter to a single white space
my @words = split(/ /,$string);

print join('--',@words),"\n";
I--do--not--like--green--eggs--and--ham
```

Notice that there are no white spaces in the printed array. The delimiter was removed.

Using qw() to create a list of words

```
my @array = ('one', 'two', 'three', 'four');
```

It is a lot of work to type all the quotes and commas.
Use qw() instead:

```
my @array = qw( one two three four );
```

qw() will produce a list of quoted words:
('one' , 'two' , 'three' , 'four')
that can now be saved as an array

Sorting

```
my @sorted_array = sort (@array)
```

The sort() function is used to sort a list. The default behavior is to sort in ascii order. A sorted list is returned.

```
my @words = qw(I do not like green eggs and ham);

my @sorted_words = sort @words;
print join('--' , @sorted_words),"\n";
```

Output

```
I--and--do--eggs--green--ham--like--not
```

ascii sort order:
0-9
A-Z
a-z

Default Sort: sort {$a cmp $b}

```
my @sorted_array = sort {$a cmp $b}(@array)
```

The sort() function performs a series of pairwise comparisons of all the elements in the list. For example it compares the first ($a) and second ($b) elements, tests if $a is less than $b, then it makes another pairwise comparison and so on until the list is sorted.

```
my @words = qw(I do not like green eggs and ham);

##sort {$a cmp $b} is default sort() behavior
my @sorted_words = sort {$a cmp $b} @words;
print join('--' , @sorted_words),"\n";
```

sort @array
is equivalent to
sort {$a cmp $b}

Output

```
I--and--do--eggs--green--ham--like--not
```

$a and $b are special Perl variables and do no need to be declared. If you use these elsewhere in the same scope, the sort function won't work. This is another reason not to use uninformative variable names (like a and b) when you're writing your scripts!

Quick Review:
The comparison operator and strings

```
my $x = 'sid';
my $y = 'nancy';
my $result = $x cmp $y;
```

$result is:

-1 if the left side is less than the right side
 0 if the left side equals the right side
+1 if the left side is greater than the right side

Quick Review:
The comparison operator and numbers

```
my $x = 2;
my $y = 3.14;
my $result = $x <=> $y;
```

$result is:

-1 if the left side is less than the right side
 0 if the left side equals the right side
+1 if the left side is greater than the right side

The comparison operator

use cmp to compare two strings

```
my $x = 'sid';
my $y = 'nancy';
my $result = $x cmp $y;
```

use <=> to compare two numbers

```
my $x = 2;
my $y = 3.14;
my $result = $x <=> $y;
```

Modify sort behavior for Numeric Sorting

The default sort can be modified by specifying the sort behavior in {} using Perl reserved variables $a and $b.

```
my @numbers = (15,2,10,20,11,1);
```

```
## default sorting is ascii
my @sorted_numbers = sort @numbers;
print "@sorted_numbers\n";
```

Output

```
1 10 11 15 2 20
```

```
## modify to sort numerically
@sorted_numbers = sort {$a <=> $b}@numbers;
print "@sorted_numbers\n";
```

Output

```
1 2 10 11 15 20
```

More sort() customization with uc()

Before $a and $b are compared they are uppercased. This only changes the temporary copy of the array elements stored in $a and $b during the sort. The actual array elements are not being changed. It is a sorted list of the original list that is being returned

```
my @sorted = sort { uc($a) cmp uc($b) } @array;
```

```
my @words = qw(I do not like green eggs and ham);
my @sorted_words = sort {uc($a) cmp uc($b)}(@words);
print join('--',@sorted_words),"\n";
```

Output

```
and--do--eggs--green--ham--I--like--not
```

The returned list is in the same format as the original list. The uc() used on $a and $b did not change the @array

Sort based on the length of each element

```
my @sorted = sort { length($a) <=> length($b) } @array;
```

```
my @words = qw(I do not like green eggs and ham);
my @sorted_words = sort {length($a) <=> length($b)}(@words);
print join('--',@sorted_words),"\n";
```

Output

```
I--do--not--and--ham--like--eggs--green
```

The returned list is in the same format as the original list. The length() used on $a and $b did not change the @array

Noticed that the <=> is used. The lengths of the words are being compared.

Warning Advanced!! Sort on length then on alphabet.

```
my @sorted_words = sort {length($a) <=> length($b) || uc($a) cmp uc($b) }(@words);
```

Reverse Sorting

Reverse the order of $a and $b to reverse the results of sort.

```
my @words = qw(I do not like green eggs and ham);
my @sorted_words = sort {$b cmp $a} @words;
print join('--',@sorted_words),"\n";
```

Output

```
not--like--ham--green--eggs--do--and--I
```

Swapping the values of 2 elements

```
my @words = qw(I do not like green eggs and ham);
print "Before Swap : w5=$words[5] w7=$words[7]\n";

my $val_1 = $words[5];
my $val_2 = $words[7];
$words[5] = $val_2;
$words[7] = $val_1;

print "After Swap  : w5=$words[5]  w7=$words[7]\n";
print join('--',@words),"\n";
```

Output

```
Before Swap : w5=eggs w7=ham
After Swap  : w5=ham  w7=eggs
I--do--not--like--green--ham--and--eggs
```

What is wrong with this?

```
$words[5] = $words[7];
$words[7] = $words[5];
print join('--',@words),"\n";
```

```
I--do--not--like--green--ham--and--ham
```

Swapping values

```
my @words = qw(I do not like green eggs and ham);
print "Before Swap : w5=$words[5] w7=$words[7]\n";";

($words[5],$words[7]) = ($words[7],$words[5]);

print "After Swap  : w5=$words[5]  w7=$words[7]\n";
print join('--',@words),"\n";
```

Output

```
Before Swap:
w5:eggs w7:ham
after swap:
w5:ham w7:eggs
I--do--not--like--green--ham--and--eggs
```

- Loops
  - foreach() : perfect for arrays
  - for() : good for arrays and much more
  - while() : perfect for many things other than arrays as well as lines of files

foreach loop

The foreach loop is especially equipped to iterate through each element of a list. It retrieves the value of each element of the list, one at at time, in the order of indices, and stores it in a variable for use within the foreach code block.

$array[0]	$array[1]	$array[2]	$array[3]	$array[4]	$array[5]
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);

foreach my $one_element(@array){
  ##do something to each $one_element


}
```

foreach loop

The foreach loop is especially equipped to iterate through each element of a list. It retrieves the value of each element of the list, one at at time, in the order of indices, and stores it in a variable for use within the foreach code block.

$array[0]	$array[1]	$array[2]	$array[3]	$array[4]	$array[5]
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);

foreach my $one_element(@array){
  ##do something to each $one_element
  print "Number: $one_element\n";
}
```

Output

```
Number: 15
Number: 2
Number: 10
Number: 20
Number: 11
Number: 1
```

A foreach loop **knows** everything about your array.

foreach() code block

All the lines within the foreach code block will be executed on each array element, one at a time.

```
my @words = qw(I do not like green eggs and ham);

foreach my $word (@words){
  my $uc_word = uc($word);
  my $len = length($word);
  print "word: $uc_word($len)\n";
}
```

Output

```
word: I(1)
word: DO(2)
word: NOT(3)
word: LIKE(4)
word: GREEN(5)
word: EGGS(4)
word: AND(3)
word: HAM(3)
```

`start at $index=0;`

1.The value of the index $index is retrieved from @words and a copy is stored in $word.
2. $word is uppercased and the result is stored in $uc_word.
3.The length of $word is calculated and stored in $len.
4. $uc_word and $length are printed.
5. Increment to the next index ($index++).
6.Go to Step **1**, repeat until the foreach code block is executed on all elements

for loop

The for loop is especially equipped to keep count and for repeating a block of code until a numerical condition is met.

```
for(initialization; test; increment){
   statements;
}
```

```
for (my $i=0; $i<5 ; $i++){
  print "\$i is: $i\n";
}
```

Output

```
$i is: 0
$i is: 1
$i is: 2
$i is: 3
$i is: 4
```

A for loop **does not know** anything about your array.

for loop

The for loop can also be used with @arrays. The $i can be used to retrieve each the value of each index.

$array[0]	$array[1]	$array[2]	$array[3]	$array[4]	$array[5]
15	2	10	20	11	1

```
my @array = (15,2,10,20,11,1);
for (my $i=0; $i<scalar @array ; $i++){
  my $value = $array[$i];
  print "value of $i is $value\n";
}
```

Output

```
value of 0 is 15
value of 1 is 2
value of 2 is 10
value of 3 is 20
value of 4 is 11
value of 5 is 1
```

Loops are similar to the steps in a thermocycler program

```
start at $i=0;
```

1. if $i is less than the length of the @array (scalar @array) then the code in the for block will be executed.
2. $value is set to contain the contents of $array[$i].
3. $value is printed
4. $i is auto incremented. ($i=$i+1);
5. Go to Step **1**, repeat as long as the test ($i<scalar @array) remains true.

Thermocycler Program and loops

Standard PCR program

1. 94 °C 3 min : Initial Denature
2. 94 °C 30 sec : Denature
3. 57 °C 30 sec : Annealing
4. 72 °C 1 min :Extension
5. Go to step 2, for additional 29 times
6. 72 °C 5 min
7. 4 °C for ever

```
my ($temp, $time);
my $cycles = 30;

($temp,$time) = (94,"3min");
doDenature($temp,$time);

for (my $i=0 ; $i<$cycles ; $i++){

  ($temp,$time) = (94, "30sec");
  doDenature($temp,$time);
  ($temp,$time) = (57, "30sec");
  doAnnealing($temp,$time);
  ($temp,$time) = (72, "1min");
  doExtension($temp,$time);
}
($temp,$time) = (72, "5min");
doAnnealing($temp,$time);

while (1){
  ($temp,$time) = (4, "forever");
  doChilling($temp,$time);
}
```

while loop

while loops continue to execute the while code block until the while conditional statement is false.

```
while (condition){
    code;
}
```

A while loop **does not know** anything about your array.

while loops and <FILEHANDLES>

while loops are great for getting lines from a file one by one and executing code on each line. It will continue until the condition is false.

```
open (IN, "<", "file.txt") or die "Can't open file.txt $!";

while (my $line = <IN>){
    chomp $line;
    print "$line\n";
}
```

Output

```
file line 1
file line 2
file line 3
```

```
start at line 1;
```

1. <> is an operator that returns the contents of one line from a file. If the end of the file is reached, nothing is returned, and nothing is false.
2. if the contents of $line is true the code block is executed.
3. $line is chomped, then printed.
4. Go to Step 1.

while loop

while loops can also be used for counting.

```
my $i = 0;
while ($i<5){
    print "\$i is $i\n";
   $i++;
}
```

This instance of the while loop functions like a for loop.

1. A counter is initialized
2. there is a test that incorporates the counter
3. the counter is changed in each iteration of the loop.

Output

```
$i is: 0
$i is: 1
$i is: 2
$i is: 3
$i is: 4
```

for and while loop can do the same thing

for and while loops can be used to do the same things, the format is just different. Neither way is better, just different

for(;;){}

```
for (my $i=0; $i<5 ; $i++){
  print "$i\n";
}
```

$i	$i<5	print "$i\n";	$i++
0	yes	0	1
1	yes	1	2
2	yes	2	3
3	yes	3	4
4	yes	4	5
5	no		

while(){}

```
my $i = 0;
while ($i<5){
    print "$i\n";
   $i++;
}
```

Different loops can do the same things

foreach and for loops with arrays

```
my @array = (15,2,10,20,11,1);
```

```
foreach my $ele(@array){
   print "$ele\n";
}
```

```
for (my $i=0; $i<scalar @array ; $i++){
  my $ele = $array[$i]
  print "$ele\n";
}
```

for and while loops with counters

```
for (my $i=0; $i<5 ; $i++){
  print "$i\n";
}
```

```
my $i = 0;
while ($i<5){
  print "$i\n";
  $i++;
}
```

Loop Control: next()

execution of next() will cause the loop to jump to the next iteration. Any code, in the loop block, that falls after the next will be skipped. The next iteration of the loop will commence. All code after the loop block will also be executed.

```
my @words = qw(I do not like green eggs and ham);

foreach my $word (sort {uc($a) cmp uc($b)}@words){
    if ($word eq 'and'){
      next;
    }
    print "$word\n";
}
```

Every element but 'and' is printed.

Output

```
do
eggs
green
ham
I
like
not
```

Loop Control: last

execution of last() will cause the loop to exit the loop. Any code, in the loop block, that falls after the last will be skipped. No further iterations will be attempted. All code that falls after the loop block will also be executed.

```
my @words = qw(I do not like green eggs and ham);

foreach my $word (@words){
    if ($word eq 'and'){
        last;
    }
    print "$word\n";
}
```

```
I
do
not
like
green
eggs
```

Every word before 'and' in @words is printed. When the element is equal to 'and' the current iteration ends, the loop block is exited and no other words are printed

Sorting, Arrays and Loops

```
my @words = qw(I do not like green eggs and ham);
#my @sorted = sort {uc($a)cmp uc($b)}@words;

foreach my $word (sort {uc($a)cmp uc($b)}@words){
    print "$word\n";
}
```

Previously, the array was sorted and the sorted results were stored in a new array

Output

```
and
do
eggs
green
ham
I
like
not
```

Here,
1. the array is sorted
2. the final sorted results are returned to the foreach loop.
3. Then one element at a time, in the sorted list will be stored in $word
4. Each element stored in $word will be processed in the loop code block

Example usage of a foreach loop

```
my @seqs = qw(TTT CGG ATG TAA CCC ACC TGA);

my $count = 0;
foreach my $seq (@seqs){
  if ($seq eq 'TAA' or $seq eq 'TGA' or $seq eq 'TAG'){
    print "*\n";
  }else {
    $count++;
  }
}
print "$count non-stop codons\n";
```

Output

```
*
*
5 non-stop codons
```

@ARGV

@ARGV is a special Perl array that automatically contains the list of arguments that follow the script name on the command line.

```
./sample_usr_input.pl 5 five
```

$ARGV[0]	$ARGV[1]
5	five

```
print "\@ARGV: @ARGV\n";
print "\$ARGV[0]: $ARGV[0]\n";
print "\$ARGV[1]: $ARGV[1]\n";

my $arg1 = shift @ARGV;
my $arg2 = shift @ARGV;


print "arg1: $arg1\n";
print "arg2: $arg2\n";
```

Output

```
@ARGV: 5 five

$ARGV[0]: 5
$ARGV[1]: five

arg1: 5
arg2: five
```

Code Example: Same analysis on multiple files

You have multiple files you want to process in the same way. Use loops!!

```
./processManyFiles.pl *txt

my @files = @ARGV;

foreach my $file (@files){
  print $file , "\n";
  open (INFILE, "<" , $file) or die "Can't open $file: $!\n";
  while (my $line = <INFILE>){
    chomp $line;
    print "\t", length $line , "\n";
  }
}
```