Perl Programming 2

Bioinformatics: Issues and Algorithms

CSE 308-408 • Fall 2007 • Lecture 5

Administrative notes

- Homework #1 is due on Tuesday, Sept. 11 at 5:00 pm. Submit your work using Blackboard Assignment function.

- Homework #2 will be available on Blackboard on Thursday, Sept. 13 at 9:00 am.

CSE Department Ice Cream Social (yum!)
Location: Packard Lab 360
Date: Tues., Sept. 11, 4:10 pm – 5:00 pm

Arrays

As we know, in bioinformatics, much of the data we care about consists of collections of genetic sequences. Simple scalar variables won't suffice ...

```
#! /usr/bin/perl -w

# The 'arrays1' program.

@list_of_sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "$list_of_sequences[1]\n";
```

A perl list data structure

Perl array variables start with "@"

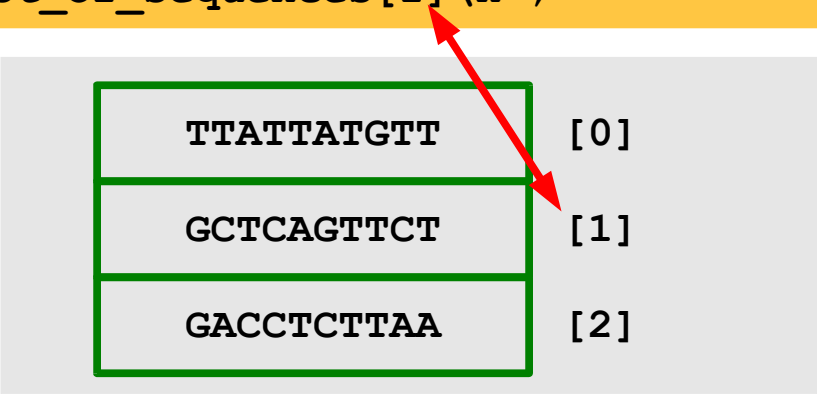
```
metis:~/CSE308/Chapter4% arrays1
GCTCAGTTCT
metis:~/CSE308/Chapter4%
```

Why did this print *GCTCAGTTCT* and not *TTATTATGTT*?

Arrays

Arrays in Perl (and many other languages) start at index [0]:

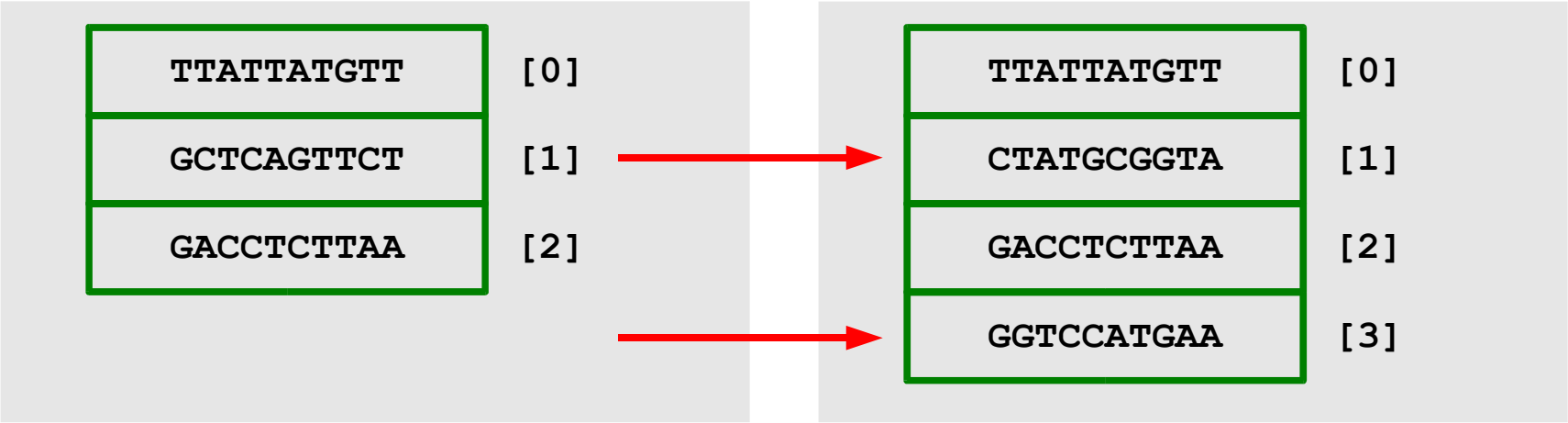
```
#! /usr/bin/perl -w

# The 'arrays1' program.

@list_of_sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "$list_of_sequences[1]\n";
```




```
metis:~/CSE308/Chapter4% arrays1
GCTCAGTTCT
metis:~/CSE308/Chapter4%
```

Manipulating arrays

```
#! /usr/bin/perl -w

# The 'arrays2' program.

@list_of_sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "$list_of_sequences[1]\n";

$list_of_sequences[1] = 'CTATGCGGTA';
$list_of_sequences[3] = 'GGTCCATGAA';

print "$list_of_sequences[1]\n";
```



Manipulating arrays

```
#! /usr/bin/perl -w

# The 'arrays2' program.

@list_of_sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "$list_of_sequences[1]\n";

$list_of_sequences[1] = 'CTATGCGGTA';
$list_of_sequences[3] = 'GGTCCATGAA';

print "$list_of_sequences[1]\n";
```

What does this do when it runs?

```
metis:~/CSE308/Chapter4% arrays2
GCTCAGTTCT
CTATGCGGTA
metis:~/CSE308/Chapter4%
```

How big is an array?

```
#! /usr/bin/perl -w

# The 'arrays3' program.

@list_of_sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "The array size is: ", $#list_of_sequences+1, ".\n";
print "The array size is: ", scalar @list_of_sequences, ".\n";
```

Returns largest array index

Perl's scalar function converts array to a scalar by counting number of list elements

```
metis:~/CSE308/Chapter4% arrays3
The array size is: 3.
The array size is: 3.
metis:~/CSE308/Chapter4%
```

Adding elements to an array

```
#! /usr/bin/perl -w

# The 'arrays4' program.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "The array size is: ", $#sequences+1, ".\n";

@sequences = ( @sequences, 'CTATGCGGTA' ) ;

print "The array size is: ", scalar @sequences, ".\n";
```

Perl combines these two lists

```
metis:~/CSE308/Chapter4% arrays4
The array size is: 3.
The array size is: 4.
metis:~/CSE308/Chapter4%
```

But be careful

Notice the effect of this code:

```
#! /usr/bin/perl -w

# The 'arrays6' program.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );

print "The array size is: ", $#sequences+1, ".\n";
print "@sequences\n";

@sequences = ( 'CTATGCGGTA' );

print "The array size is: ", scalar @sequences, ".\n";
print "@sequences\n";
```

Overwrites the array

```
metis:~/CSE308/Chapter4% arrays6
The array size is: 3.
TTATTATGTT GCTCAGTTCT GACCTCTTAA
The array size is: 1.
CTATGCGGTA
metis:~/CSE308/Chapter4%
```

Adding elements to an array

An obvious extension:

```
metis:~/CSE308/Chapter4% more arrays8
#! /usr/bin/perl -w

# The 'arrays8' program.

@sequence_1 = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA' );
@sequence_2 = ( 'GCTCAGTTCT', 'GACCTCTTAA' );
@combined_sequences = ( @sequence_1, @sequence_2 );

print "@combined_sequences\n";
metis:~/CSE308/Chapter4%
```

```
metis:~/CSE308/Chapter4% arrays8
TTATTATGTT GCTCAGTTCT GACCTCTTAA GCTCAGTTCT GACCTCTTAA
metis:~/CSE308/Chapter4%
```

Removing elements from an array: splicing

Perl provides function for “surgically removing” part of an array:

```
#! /usr/bin/perl -w

# The 'remove1' program.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA', 'TTATTATGTT' );
@removed_elements = splice @sequences, 1, 2;

print "@removed_elements\n";
print "@sequences\n";
```

Remove two array elements starting at index [1]

```
metis:~/CSE308/Chapter4% splice1
GCTCAGTTCT GACCTCTTAA
TTATTATGTT TTATTATGTT
metis:~/CSE308/Chapter4%
```

Removed elements

New array

Removing elements from an array: splicing

```
splice @sequences, OFFSET, LENGTH
```

Start removing at this array index

Remove this many elements

Notes:

- Splice subroutine returns removed elements.
- If no value for LENGTH provided, every element from OFFSET onward is removed.
- If no value for OFFSET provided, every element is removed.
- In latter case, more efficient to write `@sequences = ();`

Accessing elements in an array: slicing

To access array elements without removing them, use slice:

```
#! /usr/bin/perl -w

# The 'slices' program - slicing arrays.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );
print "@sequences\n";
@seq_slice = @sequences[ 1 .. 3 ];
print "@seq_slice\n";
print "@sequences\n";
@removed = splice @sequences, 1, 3;
print "@sequences\n";
print "@removed\n";
```

Slice to access elements 1-3

Splice to remove elements 1-3

```
europa:~/CSE308/Chapter4% slices
TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA ATCTGACCTC
GCTCAGTTCT GACCTCTTAA CTATGCGGTA
TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA ATCTGACCTC
TTATTATGTT ATCTGACCTC
GCTCAGTTCT GACCTCTTAA CTATGCGGTA
europa:~/CSE308/Chapter4%
```

Slice

Splice

Accessing elements in an array: slicing

Perl range operator

```
@dnas[ 1 .. 9 ]

@dnas [ 1, 4, 9 ]
```

Access 2nd through 10th elements in array

Access 2nd, 5th, and 10th elements in array

Notes:

- To access list of elements from array, use a slice.
- To remove list of elements from array, use splice.
- Both return the elements in question.

Pushing, popping, shifting, and unshifting

Often, manipulation of arrays involves single elements, so Perl provides special functions to make this easier:

shift	Removes and returns first element from array
pop	Removes and returns last element from array
unshift	Adds element (or list) onto start of array
push	Adds element (or list) onto end of array

Start of array

```
@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );
```

End of array

Pushing, popping, shifting, and unshifting

```
#! /usr/bin/perl -w

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );

print "@sequences\n";
$last = pop @sequences;
print "@sequences\n";
$first = shift @sequences;
print "@sequences\n";
unshift @sequences, $last;
print "@sequences\n";
push @sequences, ( $first, $last );
print "@sequences\n";
```

#1 Removes last element

#2 Removes first element

#3 Places element at start

#4 Places elements at end

```
europa:~/CSE308/Chapter4% pushpop
TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA ATCTGACCTC
TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA
GCTCAGTTCT GACCTCTTAA CTATGCGGTA
ATCTGACCTC GCTCAGTTCT GACCTCTTAA CTATGCGGTA
ATCTGACCTC GCTCAGTTCT GACCTCTTAA CTATGCGGTA TTATTATGTT
ATCTGACCTC
europa:~/CSE308/Chapter4%
```

#1

#2

#3

#4

Pushing, popping, shifting, and unshifting

TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA ATCTGACCTC

pop last element (**ATCTGACCTC**)

TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA

"pop"
$last

TTATTATGTT GCTCAGTTCT GACCTCTTAA CTATGCGGTA

"shift"
$first

shift element (**TTATTATGTT**)

GCTCAGTTCT GACCTCTTAA CTATGCGGTA

$last
"unshift"

GCTCAGTTCT GACCTCTTAA CTATGCGGTA

unshift one new element (**ATCTGACCTC**)

ATCTGACCTC GCTCAGTTCT GACCTCTTAA CTATGCGGTA

ATCTGACCTC GCTCAGTTCT GACCTCTTAA CTATGCGGTA

push on two new elements (**TTATTATGTT ATCTGACCTC**)

ATCTGACCTC GCTCAGTTCT GACCTCTTAA CTATGCGGTA TTATTATGTT ATCTGACCTC

$first, $last
"push"

Iterating over all elements of an array

Perl makes it easy to iterate over all the elements of an array:

```
#! /usr/bin/perl -w
# The 'iterateW' program - iterate over an entire array.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );

$index = 0;
$last_index = $#sequences;

while ( $index <= $last_index )
{
        print "$sequences[ $index ]\n";
        ++$index;
}
```

```
phoebe:~/CSE308/Chapter4% iterateW
TTATTATGTT
GCTCAGTTCT
GACCTCTTAA
CTATGCGGTA
ATCTGACCTC
phoebe:~/CSE308/Chapter4%
```

Iterating over all elements of an array, take 2

Perl also provides an even easier way to do this:

```
#! /usr/bin/perl -w

# The 'iterateF' program - iterate over an entire array
# with 'foreach'.

@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );

foreach $value ( @sequences )
{
        print "$value\n";
}
```

Step through all elements

Note: changing scalar $value also changes array!

```
phoebe:~/CSE308/Chapter4% iterateF
TTATTATGTT
GCTCAGTTCT
GACCTCTTAA
CTATGCGGTA
ATCTGACCTC
phoebe:~/CSE308/Chapter4%
```

Easier list representations

Lists in Perl are comma-separated collections of scalars.

They can be represented in a number of ways, however:

```
@sequences = ( 'TTATTATGTT', 'GCTCAGTTCT', 'GACCTCTTAA',
               'CTATGCGGTA', 'ATCTGACCTC' );
```

```
@sequences = ( TTATTATGTT, GCTCAGTTCT, GACCTCTTAA,
               CTATGCGGTA, ATCTGACCTC );
```

You don't need quotes if there aren't any spaces

```
@sequences = qw( TTATTATGTT GCTCAGTTCT GACCTCTTAA
                 CTATGCGGTA ATCTGACCTC );
```

Can eliminate commas by using “qw” (“quote words”)

Hashes

In addition to arrays, Perl provides *hashes* , another powerful data structure that will come in handy on many occasions.

[0]	TTATTATGTT
[1]	GCTCAGTTCT
[2]	GACCTCTTAA

Perl array:
indexing is implicit

seqA	TTATTATGTT
seqZ	GCTCAGTTCT
seqC	GACCTCTTAA

Perl hash:
element accessed by specifying value
("associative array")

Use "%" to indicate hash

```
%sequence_hash = ( seqA, TTATTATGTT, seqZ, GCTCAGTTCT,
                   seqC, GACCTCTTAA)
```

Working with hashes

```
#! /usr/bin/perl -w
# The 'hash1' program.

%nucleotide_bases = ( A, Adenine, T, Thymine );

print "The expanded name for 'A' is $nucleotide_bases{ 'A' }\n";
```

To access a hash element, refer to its name

```
phoebe:~/CSE308/Chapter4% hash1
The expanded name for 'A' is Adenine
phoebe:~/CSE308/Chapter4%
```

```
#! /usr/bin/perl -w
# The 'hash2' program.

%nucleotide_bases = ( A, Adenine, T, Thymine );
@hash_names = keys %nucleotide_bases;

print "The names in the %nucleotide_bases hash are: @hash_names\n";
```

To determine names for a hash, use keys function

```
phoebe:~/CSE308/Chapter4% hash2
The names in the %nucleotide_bases hash are: A T
phoebe:~/CSE308/Chapter4%
```

Working with hashes

```
#! /usr/bin/perl -w
# The 'hash3' program.

%nucleotide_bases = ( A, Adenine, T, Thymine );
$hash_size = keys %nucleotide_bases;

print "The size of the %nucleotide_bases hash is: $hash_size\n";
```

To determine size of a hash, use a scalar context

```
phoebe:~/CSE308/Chapter4% hash3
The size of the %nucleotide_bases hash is: 2
phoebe:~/CSE308/Chapter4%
```

To add entries to an existing hash, do this:

```
%nucleotide_bases = ( A, Adenine, T, Thymine );
...
$nucleotide_bases{ 'G' } = 'Guanine';
$nucleotide_bases{ 'C' } = 'Cytosine';
```

Working with hashes

```
#! /usr/bin/perl -w

# The 'hash4' program.

%nucleotide_bases = ( A, Adenine, T, Thymine );

$nucleotide_bases{ 'G' } = 'Guanine';
$nucleotide_bases{ 'C' } = 'Cytosine';

@hash_keys = keys %nucleotide_bases;
$hash_size = keys %nucleotide_bases;

print "The keys of the %nucleotide_bases hash are @hash_keys\n";
print "The size of the %nucleotide_bases hash is: $hash_size\n";
```

Note: Perl does not store hashes in insertion order!

```
phoebe:~/CSE308/Chapter4% hash4
The keys of the %nucleotide_bases hash are A T C G
The size of the %nucleotide_bases hash is: 4
phoebe:~/CSE308/Chapter4%
```

Moral: don't count on internal ordering of hash elements.

Working with hashes

As a more readable shorthand notation for this:

```
%nucleotide_bases = ( A, Adenine, T, Thymine,
                      G, Guanine, C, Cytosine );
```

Perl lets you do this:

```
%nucleotide_bases = ( A => Adenine, T => Thymine,
                      G => Guanine, C => Cytosine );
```

You may use “=>” wherever you'd use a comma, although some places are obviously better than others ...

Removing entries from a hash

```
#! /usr/bin/perl -w
# The 'hash5' program.

%nucleotide_bases = ( A => Adenine, T => Thymine,
        G => Guanine, C => Cytosine );

@hash_keys = keys %nucleotide_bases;

print "The keys of the %nucleotide_bases hash are @hash_keys\n";

delete $nucleotide_bases{ 'G' };

@hash_keys = keys %nucleotide_bases;

print "The keys of the %nucleotide_bases hash are @hash_keys\n";
```

Removes both name and value

```
phoebe:~/CSE308/Chapter4% hash5
The keys of the %nucleotide_bases hash are A T C G
The keys of the %nucleotide_bases hash are A T C
phoebe:~/CSE308/Chapter4%
```

Undefining variables

```
#! /usr/bin/perl -w
# The 'hash6' program.

%nucleotide_bases = ( A => Adenine, T => Thymine,
        G => Guanine, C => Cytosine );

@hash_keys = keys %nucleotide_bases;

print "The keys of the %nucleotide_bases hash are @hash_keys\n";

$nucleotide_bases{ 'G' }  = undef;

@hash_keys = keys %nucleotide_bases;

print "The keys of the %nucleotide_bases hash are @hash_keys\n";
print "The expanded name for 'G' is $nucleotide_bases{ 'G' }\n";
```

This hash entry still exists, but its value is undefined

```
phoebe:~/CSE308/Chapter4% hash6
The keys of the %nucleotide_bases hash are A T C G
The keys of the %nucleotide_bases hash are A T C G
Use of uninitialized value in concatenation (.) or string
at ./hash6 line 17.
The expanded name for 'G' is
phoebe:~/CSE308/Chapter4%
```

Perl complains when you try to use undefined variable

Slicing hashes

```
#! /usr/bin/perl -w

# The 'hash7' program.

%gene_counts = ( Human                  => 31000,
                 'Thale cress'          => 26000,
                 'Nematode worm'        => 18000,
                 'Fruit fly'            => 13000,
                 Yeast                  => 6000,
                 'Tuberculosis microbe' => 4000 );

@counts = @gene_counts{ Human, 'Fruit fly', 'Tuberculosis microbe' };

print "@counts\n";
```

Note selective use of single quote character

Hash slice

Good formatting makes this easy to read

```
phoebe:~/CSE308/Chapter4% hash7
31000 13000 4000
phoebe:~/CSE308/Chapter4%
```

Note this is an array of values

A complete example

```
#! /usr/bin/perl -w
# The 'genes' program - a hash of gene counts.

use constant    LINE_LENGTH => 60;

%gene_counts = ( Human                    => 31000,
                 'Thale cress'            => 26000,
                 'Nematode worm'          => 18000,
                 'Fruit fly'              => 13000,
                 Yeast                    => 6000,
                 'Tuberculosis microbe'   => 4000 );

print '-' x LINE_LENGTH, "\n";

while ( ( $genome, $count ) = each %gene_counts )
{
        print "'$genome' has a gene count of $count\n";
}

print '-' x LINE_LENGTH, "\n";

foreach $genome ( sort keys %gene_counts )
{
        print "'$genome' has a gene count of $gene_counts{ $genome }\n";
}

print '-' x LINE_LENGTH, "\n";
```

Perl repetition operator (x)

Returns successive name/value pairings

Steps through sorted keys

A complete example

```
phoebe:~/CSE308/Chapter4% genes
-----------------------------------------------------------
'Human' has a gene count of 31000
'Tuberculosis microbe' has a gene count of 4000
'Fruit fly' has a gene count of 13000
'Nematode worm' has a gene count of 18000
'Yeast' has a gene count of 6000
'Thale cress' has a gene count of 26000
-----------------------------------------------------------
'Fruit fly' has a gene count of 13000
'Human' has a gene count of 31000
'Nematode worm' has a gene count of 18000
'Thale cress' has a gene count of 26000
'Tuberculosis microbe' has a gene count of 4000
'Yeast' has a gene count of 6000
-----------------------------------------------------------
phoebe:~/CSE308/Chapter4%
```

Note that keys are sorted here

Maxims from BBP Chapter 4

More key points to ponder as you start to program in Perl:

- Lists in Perl are comma-separated collections of scalars.
- Perl starts counting from zero, not one.
- Three main contexts in Perl: numeric, list, and scalar.
- To access list of values from array, use a slice.
- To remove list of values from array, use **`splice`**.
- Use **`foreach`** to process every element in an array.
- A hash is a collection of name / value pairings.
- Hash name parts must be unique.

Intro to subroutines

Following line was repeated 3 times in our complete example:

```
print '-' x LINE_LENGTH, "\n";
```

"Print a dash character LINE_LENGTH times and then follow this by printing a newline."

Seems kind of cryptic and not very general ...

... wouldn't it be nice to replace it by something more like this:

```
drawline "-", LINE_LENGTH;
```

"Draw a line of dashes LINE_LENGTH long."

Or this:

```
drawline( "-", LINE_LENGTH );
```

Intro to subroutines

```
#! /usr/bin/perl -w

# first_drawline - the first demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
        print "-" x REPEAT_COUNT, "\n";
}

print "This is the first_drawline program.\n";
drawline;
print "It's purpose is to demonstrate the first version of drawline.\n";
drawline;
print "Sorry, but it is not very exciting.\n";
```

Subroutine "drawline" specified here

Subroutine invoked here

```
phoebe:~/CSE308/Chapter5% first_drawline
This is the first_drawline program.
------------------------------------------------------------
It's purpose is to demonstrate the first version of drawline.
------------------------------------------------------------
Sorry, but it is not very exciting.
phoebe:~/CSE308/Chapter5%
```

Better, more flexible subroutines

Our previous example was quite limited:

```
sub drawline {
        print "-" x REPEAT_COUNT, "\n";
}
```

- Only prints dash (-) character.
- Only prints character REPEAT_COUNT times.

Subroutines can accept parameters as input:

```
drawline "-", LINE_LENGTH;
```

"Draw a line consisting of the specified character of the specified length."

Better, more flexible subroutines

Subroutines can accept parameters as input:

```
drawline "-", LINE_LENGTH;
```

First parameter (character to use)

Second parameter (line length)

```
sub drawline {
        print $_[0] x $_[1], "\n";
}
```

@_ is called "default array"

(This notation works, but it's a little awkward. We'll see something better soon.)

A better drawline subroutine

```
#! /usr/bin/perl -w
# second_drawline - the second demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
        print $_[0] x $_[1]  "\n";
}

print "This is the second_drawline program.\n";
drawline "-", REPEAT_COUNT;
print "Sorry, but it is still not very exciting. However, it is more useful.\n";

drawline "=", REPEAT_COUNT;
drawline "-oOo-", 12;
drawline "- ", 30;
drawline ">>==<<==", 8;
```

First parameter

Second parameter

Note variety of ways drawline can be invoked

```
altair:~/CSE308/Chapter5% second_drawline
This is the second_drawline program.
------------------------------------------------------------
Sorry, but it is still not very exciting. However, it is more useful.
============================================================
-oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo-
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==
altair:~/CSE308/Chapter5%
```

Using shift() to process the default array

```
#! /usr/bin/perl -w
# third_drawline - the third demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
        print shift() x shift(), "\n";
}

print "This is the third_drawline program.\n";
drawline "-", REPEAT_COUNT;
print "Sorry, but it is still not very exciting. However, it is more useful.\n";

drawline "=", REPEAT_COUNT;
drawline "-oOo-", 12;
drawline "- ", 30;
drawline ">>==<<==", 8;
```

Each call to shift() returns next item in default array

```
europa:~/CSE308/Chapter5% third_drawline
This is the third_drawline program.
------------------------------------------------------------
Sorry, but it is still not very exciting. However, it is more useful.
============================================================
-oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo-
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==>>==<<==
europa:~/CSE308/Chapter5%
```

Better processing of parameters

What happens if we call a subroutine with too few parameters?

```
#! /usr/bin/perl -w
# third_drawline - the third demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
        print shift() x shift(), "\n";
}

print "This is the third_drawline program.\n";
drawline;
print "Sorry, but it is still not very exciting. However, it is more useful.\n";

drawline "=", REPEAT_COUNT;
drawline "-oOo-", 12;
drawline "- ", 30;
drawline ">>==<<==", 8;
```

Note missing parameters

```
europa:~/CSE308/Chapter5% third_drawline
This is the third_drawline program.
Use of uninitialized value in repeat (x) at ./third_drawline line 8.

...
```

It would be better if there was a reasonable default behavior here

Better processing of parameters

```
#! /usr/bin/perl -w
# fourth_drawline - the fourth demonstration program for "drawline".

use constant REPEAT_COUNT => 60;

sub drawline {
        $chars = shift || "-";
        $count = shift || REPEAT_COUNT;

        print $chars x $count, "\n";
}

print "This is the fourth_drawline program.\n";
drawline;
print "Sorry, but it is still not very exciting. However, it is more useful.\n";

drawline "=", REPEAT_COUNT;
drawline "-oOo-", 12;
drawline "- ", 30;
drawline ">>==<<==", 8;
```

If no parameters present, uses dash as default

If count not present, uses REPEAT_COUNT as default

```
europa:~/CSE308/Chapter5% fourth_drawline
This is the fourth_drawline program.
------------------------------------------------------------
Sorry, but it is still not very exciting. However, it is more useful.
============================================================
-oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo--oOo-
...
```

Better processing of parameters

"What if" scenarios ...

```
drawline "===", 10;
```

```
==============================
europa:~/CSE308/Chapter5%
```

```
drawline "*";
```

```
**************************************
europa:~/CSE308/Chapter5%
```

It would be great if drawline could handle its parameters in any order

```
drawline 40;
```

```
4040404040404040404040404040404040404040404040404040404040404040404040404040
4040404040404040404040404040404040404040404040
europa:~/CSE308/Chapter5%
```

Not what we intended!

```
drawline 20, "-";
```

```
Argument "-" isn't numeric in repeat (x) at ./fourth_drawline line 11.
europa:~/CSE308/Chapter5%
```

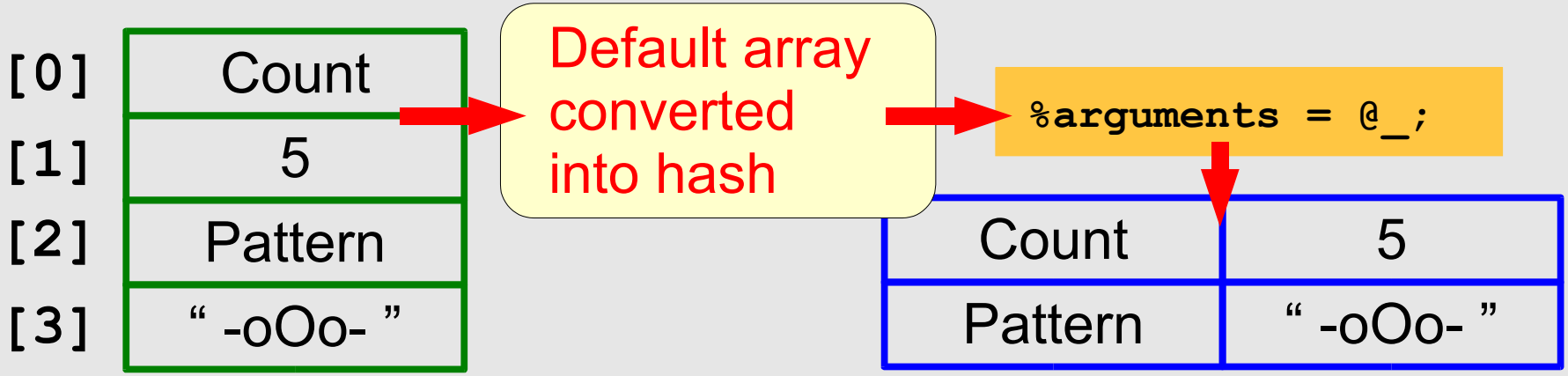
Even better processing of parameters

Perl allows programmer to pass parameters in any order:

```
drawline( Pattern => "*" );
drawline( Count => 20 );
drawline( Count => 5, Pattern => " -oOo- ");
drawline( Pattern => "===", Count => 10 );
drawline;
```

Note, however, that programmer must now also provide name

```
drawline( Count => 5, Pattern => " -oOo- ");
```



Even better processing of parameters

```
sub drawline {
        %arguments = @_;

        $chars = $arguments{ Pattern } || "-";
        $count = $arguments{ Count } || REPEAT_COUNT;

        print $chars x $count, "\n";
}
```

Convert default array to hash, then access parameters via names

```
drawline( Pattern => "*" );
drawline( Count => 20 );
drawline( Count => 5, Pattern => " -oOo- ");
drawline( Pattern => "===", Count => 10 );
drawline;
```

```
**********************************************************
-------------------
 -oOo-  -oOo-  -oOo-  -oOo-  -oOo-
==============================
------------------------------------------------------------
europa:~/CSE308/Chapter5%
```

A more flexible approach

The fact that drawline outputs a newline each time is limiting. Say we want to produce the following output:

```
+---------------+
|               |
|               |
|               |
+---------------+
```

Writing this:

```
print "+";
drawline( Count => 15 );
print "+";
```

Results in this:

Not what we want!

```
+---------------
+europa:~/CSE308/Chapter5%
```

A more flexible approach

Solve part of the problem by removing newline from drawline.

The following code fragment works then:

```
print "+";
drawline( Count => 15 );
print "+\n";
```

```
+---------------+
europa:~/CSE308/Chapter5%
```

Getting a little too ambitious, however, results in this:

```
print "+", drawline( Count => 15 ), "+\n";
```

```
---------------+1+
europa:~/CSE308/Chapter5%
```

Perl invokes drawline subroutine <u>before</u> producing any output

Return value from drawline

A more flexible approach

Even better: separate tasks of formatting and printing:

```
sub drawline {
        %arguments = @_;

        $chars = $arguments{ Pattern } || "-";
        $count = $arguments{ Count } || REPEAT_COUNT;

        return $chars x $count;
}
```

Later invocations print lines to generate a box

```
print "+", drawline, "+\n";
print "|", drawline ( Pattern => " " ), "|\n";
print "|", drawline ( Pattern => " " ), "|\n";
print "|", drawline ( Pattern => " " ), "|\n";
print "+", drawline, "+\n";
```

```
europa:~/CSE308/Chapter5% boxes
+---------------+
|               |
|               |
|               |
+---------------+
europa:~/CSE308/Chapter5%
```

Visibility and scope

Consider the following simple Perl program:

```
#! /usr/bin/perl -w
# global_scope - the effect of "global" variables.

sub adjust_up {
        $other_count = 1;
        print "count at start of adjust_up: $count\n";
        $count++;
        print "count at end of adjust_up: $count\n";
}

$count = 10;
print "count in main: $count\n";
adjust_up;
print "count in main: $count\n";
print "other_count in main: $other_count\n";
```

In other words, Perl variables are "global."

```
europa:~/CSE308/Chapter5% global_scope
count in main: 10
count at start of adjust_up: 10
count at end of adjust_up: 11
count in main: 11
other_count in main: 1
europa:~/CSE308/Chapter5%
```

By default, variables in Perl are accessible <u>anywhere</u>, no matter where they are defined.

Private variables in Perl

There are times when global accessibility is not what you want.

```
#! /usr/bin/perl -w
# private_scope - the effect of "my" variables.

sub adjust_up {
        my $other_count = 1;
        print "count at start of adjust_up: $count\n";
        $count++;
        print "count at end of adjust_up: $count\n";
}

my $count = 10;
print "count in main: $count\n";
adjust_up;
print "count in main: $count\n";
print "other_count in main: $other_count\n";
```

To declare a varaible private, use “my”

```
europa:~/CSE308/Chapter5% private_scope
count in main: 10
count at start of adjust_up:
count at end of adjust_up: 1
count in main: 10
other_count in main:
europa:~/CSE308/Chapter5%
```

adjust_up can't see count

increment of count not visible

main can't see other_count

Maxims from BBP Chapter 5

Yet more key points to keep in mind as you learn Perl:

- Whenever you think you will reuse code, create a subroutine.
- When determining scope of a variable, consider its visibility.
- Unless good reason not to, always declare variables with **`my`**.
- If you must use a global variable, declare it with **`our`**.

Debugging advice

A wise and famous saying I once encountered:

"One bug is easy to find.
Many bugs will blow your mind."

Moral: write your programs in small pieces. Thoroughly test each piece before moving on. Do <u>not</u> type in dozens of lines of Perl code and then run it, expecting it to work – it won't.

Tracking down and fixing a single bug is doable. A program that contains multiple bugs is usually beyond hope.

Wrap-up

Readings for next time:

- BB&P Chapters 6-8 (more Perl programming).

Remember:

- Come to class having done the readings.
- Check Blackboard regularly for updates.