Accessing array elements

- Accessing array elements in Perl is syntactically similar to C.
- Perhaps somewhat counterintuitively, you use `$a[<num>]` to specify a scalar element of an array named `@a`.
- The index `<num>` is evaluated as a numeric expression.
- By default, the first index in an array is 0.

Examples of arracy access

```
$a[0] = 1;          # assign numeric constant
$a[1] = "string";   # assign string constant
print $m[$a];       # access via variable
$a[$c] = $b[$d];    # copy elements
$a[$i] = $b[$i];    #
$a[$i+$j] = 0;      # expressions are okay
$a[$i]++;           # increment element
```

Assign list literals

You can assign a list literal to an array or to a list of scalars:

```
($a, $b, $c) = (1, 2, 3);          # $a = 1, $b = 2, $c = 3
($m, $n) = ($n, $m);               # works!
@nums = (1..10);                   # $nums[0]=1, $nums[1]=2, ...
($x,$y,$z) = (1,2)                 # $x=1, $y=2, $z is undef
@t = ();                           # t is defined with no elements
($a[1],$a[0])=($a[0],$a[1]);       # swap works!
@kudomono = ('apple','orange');    # list with 2 elements
@kudomono = qw/ apple orange /;    # ditto
```

Array-wide access

Sometimes you can do an operation on an entire array. Use the `@array` name:

```
@x = @y;           # copy array y to x
@y = 1..1000;      # parentheses are not requisite
@lines = <STDIN>   # very useful!
print @lines;      # works in Perl 5, not 4
```

Printing entire arrays

- If an array is simply printed, it comes out something like

```
@a = ('a','b','c','d');
print @a;
abcd
```

- If an array is interpolated in a string, you get spaces: @a = ('a','b','c','d'); print "@a"; a b c d

Arrays in a scalar context

Generally, if you specify an array in a scalar context, the value returned is the number of elements in the array.

```
@array1 = ('a', 3, 'b', 4, 'c', 5);  # assign array1 the values of list
@array2 = @array1;                   # assign array2 the values of array1
$m = @array2;                        # $m now has value 6
$n = $m + @array1                    # $n now has value 12
```

Using a scalar in an array context

If you assign an array a scalar value, that array will be just a one element array:

```
$m = 1;
@arr = $m;          # @arr == ( 1 );
@yup = "apple";     # @yup == ( "apple" );
@arr = ( undef ); # @arr == ( undef );
@arr = ();          # @arr is now empty, not an array with one undef value!
```

Size of arrays

Perl arrays can be any size up to the amount of memory available for the process. The number of elements can vary during execution.

```
my @fruit;              # has zero elements
$fruit[0] = "apple";    # now has one element
$fruit[1] = "orange";   # now has two elements
$frist[99] = 'plum';    # now has 100 elements, most of which are undef
```

Last element index

Perl has a special scalar form `$#arrayname` that returns a scalar value that is equal to the index of the last element in the array.

```
for($i = 0; $i<=$#arr1; $i++)
{
  print "$arr1[$i]\n";
}
```

Last element index use

You can also use this special scalar form to truncate an array:

```
@arr = (1..100);    # arr has 100 elements...
$#arr = 9;          # now it has 10
print "@arr";
1 2 3 4 5 6 7 8 9 10
```

Using negative array indices

A negative array index is treated as being relative to the end of the array:

```
@arr = 1..100;
print $arr[-1];    # similar to using $arr[$#arr]
100
print $arr[-2];
99
```

Arrays as stacks

- Arrays can be used as stacks, and Perl has built-ins that are useful for manipulating arrays as stacks: `push`, `pop`, `shift`, and `unshift`.

- `push` takes two arguments: an array to push onto, and what is to pushed on. If the new elment is an array, then the elements of that array are appended to the original array as scalars.

- A `push` puts the new element(s) at the end of the original array.

- A `pop` removes the last element from the array specified.

Examples of `push` and `pop`

```
push @nums, $i;
push @ans, "yes";
push @a, 1..5;
push @a, @b;          # appends the elements of b to a
push @a, (1, 3, 5);
pop @a;
push(@a,pop(@b));     # moves the last element of b to end of a
@a = (); @b = (); push(@b,pop(@a))  # b now has one undef value
```

`shift` and `unshift`

- `shift` removes the first element from an array
- `unshift` inserts an element at the beginning of an array

Examples of `shift` and `unshift`

```
@a = 1..10;
unshift @a,99;          # now @a == (99,1,2,3,4,5,6,7,8,9)
unshift @a,('a','b') # now @a == ('a','b',99,1,2,3,4,5,6,7,8,9)
$x = shift @a;          # now $x == 'a'
```

`foreach` control structure

You can use `foreach` to process each element of an array or list.

It follows the form:

```
foreach $SCAlAR (@ARRAY or LIST)
{
  <statement list>
}
```

(You can also `map` for similar processing.)

`foreach` examples

```
foreach $a (@a)
{
  print "$a\n";
}
map {print "$_\n";} @a;

foreach $item (qw/ apple pear lemon /)
{
  push @fruits,$item;
}
map {push @fruits, $_} qw/ apple pear lemon/;
```

The default variable $_

$_ is the default variable (and is used in the previous `map()` examples). It is used as a default when at various times, such as when reading input, writing output, and in the `foreach` and `map` constructions.

The default variable $_

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

$sum = 0;
foreach(@arr)
{
  $sum += $_;
}

map { $sum += $_} @arr;
```

Input from the "diamond" operator

Reading from <> causes a program to read from the files specified on the command line or stdin if no files are specified.

Example of diamond operator

```
#!/usr/bin/perl -w
# 2006 09 22 - rdl script23.pl
while(<>)
{
    print;
}
```

You can either use `./Script23.pl < /etc/hosts` or `./Script23.pl /etc/hosts /etc/resolv.conf`.

The `@ARGV` array

There is a builtin array called `@ARGV` which contains the command lines arguments passed in by the calling program.

Note that `$ARGV[0]` is the first argument, not the name of the Perl program being invoked