Arrays (Lists)

An **array** is a sequence of scalars, indexed by position (0, 1, 2, ...)

The whole array is denoted by \( @\text{array} \)

Individual array elements are denoted by \( @\text{array}[\text{index}] \)

\( @\text{array} \) gives the *index of the last element*.

Example:

```plaintext
$a[0] = "first string";
$a[1] = "2nd string";
$a[2] = 123;

# or, equivalently,
@a = ("first string", "2nd string", 123);

print "Index of last element is @a\n";
print "Number of elements is ", @a+1, "\n";
```
Arrays (Lists)

@a = ("abc", 123, 'x');

# scalar context ... gives list length
$n = @a; # $n == 3

# string context ... gives space-separated elems
$s = "@a"; # $s eq "abc 123 x"

# scalar context ... gives list length
$t = @a.""; # $t eq "3"

# list context ... gives joined elems
print @a; # displays "abc123x"

In Perl, interpretation is context-dependent.
Arrays (Lists)

Arrays do not need to be declared, and they grow and shrink as needed.

“Missing” elements are interpolated, e.g.

```perl
$abc[0] = "abc"; $abc[2] = "xyz";
# reference to $abc[1] returns ""
```

Can assign to a whole array; can assign from a whole array, e.g.

```perl
@numbers = (4, 12, 5, 7, 2, 9);
($a, $b, $c, $d) = @numbers;
```

Since assignment of list elements happens in parallel …

```perl
($x, $y) = ($y, $x);  # swaps values of $x, $y
```
Array slices, e.g.

```perl
@list = (1, 3, 5, 7, 9);
print "@list[0,2]\n";  # displays "1 5"
print "@list[0..2]\n";  # displays "1 3 5"
print "@list[4,2,3]\n";  # displays "9 5 7"
print "@list[0..9]\n";  # displays "1 3 5 7 9"
```

Array values interpolated into array literals:

```perl
@a = (3, 5, 7);
@b = @a;        # @b = (3,5,7);
@c = (1, @a, 9); # @c = (1,3,5,7,9);
@a == (@a) == ((@a)) ...
```
Arrays (Lists)

Arrays can be accessed element-at-a-time using the `for` loop:

```perl
@nums = (23, 95, 33, 42, 17, 87);

@nums = (23, 95, 33, 42, 17, 87);

$sum = 0;
# @nums in scalar context gives length
for ($i = 0; $i < @nums; $i++) {
    $sum += $nums[$i];
}

$sum = 0;
foreach $num (@nums) {
    sum += $num;
}
```
Example - /bin/echo in Perl

```perl
foreach $arg (@ARGV) {
    print $arg, " ";
}
print "\n";
```

(source code for echo.0.pl)

or using array interpolation

```perl
print "@ARGV\n";
```

(source code for echo.1.pl)

or using array interpolation

```perl
print join(" ", @ARGV), "\n";
```

(source code for echo.2.pl)
Example - Sum Command-line Arguments

```
# sum integers supplied as command line arguments
# no check that arguments are numeric
$sum = 0;
foreach $arg (@ARGV) {
    $sum += $arg;
}
print "Sum of the numbers is $sum\n";
```

source code for sum_arguments.pl
while (1) {
    print "Enter array index: ";
    $n = <STDIN>;
    if (!$n) {
        last;
    }
    chomp $n;
    $a[$n] = 42;
    print "Array element $n now contains $a[$n]\n";
    printf "Array size is now %d\n", $#a+1;
}

source code for array_growth_demo.pl
Example - Counting Lines on stdin

# Count the number of lines on standard input.
$line_count = 0;
while (1) {
    $line = <STDIN>;
    last if !$line;
    $line_count++;
}
print "$line_count lines\n";

# slightly more concise
$line_count = 0;
while (<STDIN>) {
    $line_count++;
}
print "$line_count lines\n";
# using a backwards while to be really concise

```perl
$line_count = 0;
$line_count++ while <STDIN>;
print "\$line_count lines\n";
```

source code for line_count.2.pl

# read the input into an array and use the array size.

```perl
@lines = <STDIN>;
print \$#lines+1, " lines\n";
```

source code for line_count.3.pl

# Assignment to () forces a list context.
# Hence all lines of input are read.
# The special variable $. contains the current line number

```perl
() = <STDIN>;
print "\$. lines\n";
```

source code for line_count.4.pl
push and pop act on the “right-hand” end of an array:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@a = (1,3,5);</td>
<td># Value of @a</td>
</tr>
<tr>
<td>push @a, 7;</td>
<td># (1,3,5,7)</td>
</tr>
<tr>
<td>$x = pop @a;</td>
<td># (1,3,5,7), $x == 7</td>
</tr>
<tr>
<td>$y = pop @a;</td>
<td># (1,3,5), $y == 5</td>
</tr>
</tbody>
</table>
Example - tac

# Print lines read from stdin in reverse order.
# In a C-style

while ($line = <STDIN>) {
    $line[$line_number++] = $line;
}

for ($line_number = $#line; $line_number >= 0 ; $line_number--) {
    print $line[$line_number];
}

source code for reverse_lines.0.pl

# Using <> in a list context

@line = <STDIN>;

for ($line_number = $#line; $line_number >= 0 ; $line_number--) {
    print $line[$line_number];
}

source code for reverse_lines.1.pl
# Using <> in a list context & reverse

```perl
@lines = <STDIN>;
print reverse @lines;
```

source code for reverse_lines.2.pl

# Using <> in a list context & reverse

```perl
print reverse <STDIN>;
```

source code for reverse_lines.3.pl

# Using push & pop

```perl
while ($line = <STDIN>) {
    push @lines, $line;
}
while (@lines) {
    my $line = pop @lines;
    print $line;
}
```

source code for reverse_lines.4.pl
Example - tac

```perl
# More succinctly with pop
@lines = <STDIN>;
while (@lines) {
    print pop @lines;
}

source code for reverse_lines.5.pl

# Using unshift
while ($line = <STDIN>) {
    unshift @lines, $line;
}
print @lines;

source code for reverse_lines.6.pl
```
# Simple cp implementation using line by line I/O

die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;

(infile = shift @ARGV;
(outfile = shift @ARGV;

open my $in, '<', $infile or die "Cannot open $infile: $!";
open my $out, '>', $outfile or die "Cannot open $outfile: $!";

while ($line = <$in>) {
    print $out $line;
}

close $in;

close $out;

exit 0;

source code for cp.0.pl
# Simple cp implementation using line by line I/O
# relying on the default variable 

```
die "Usage: $0 <infile> <outfile>

if @ARGV != 2;
    $infile = shift @ARGV;
    $outfile = shift @ARGV;

open my $in, '<', $infile or die "Cannot open $infile: $!";
open my $out, '>', $outfile or die "Cannot open $outfile: $!";

# loop could also be written in one line:
# print OUT while <IN>;

while (<$in>) {
    print $out;
}

close $in;

close $out;

exit 0;
```
Simple cp implementation reading entire file into array
# note that <> returns an array of lines
# in a list context (in a scalar context it returns a single line)
die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;
(infile = shift @ARGV;
(outfile = shift @ARGV;
open my $in, '<', $infile or die "Cannot open $infile: $!";
@lines = <$in>;
close $in;
open my $out, '>', $outfile or die "Cannot open $outfile: $!";
print $out @lines;
close $out;
exit 0;
# Simple cp implementation via system!
# Will break if filenames contain single quotes

die "Usage: $0 <infile> <outfile>\n" if @ARGV != 2;
(infile = shift @ARGV;
$outfile = shift @ARGV;
exit system "/bin/cp '$infile' '$outfile'";
# $/ contains the line separator for Perl
# if it is undefined we can slurp an entire file
# into a scalar variable with a single read

die "Usage: cp <infile> <outfile>\n" if @ARGV != 2;
(infile = shift @ARGV;
$outfile = shift @ARGV;
undef $/;
open my $in, '<', $infile or die "Cannot open $infile: $!";
$contents = <$in>;
close $in;
open my $out, '>', $outfile or die "Cannot open $outfile: $!";
print $out $contents;
close $out;
exit 0;

source code for cp.4.pl
Other useful operations on arrays:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@b = sort @a</td>
<td>returns sorted version of @a</td>
</tr>
<tr>
<td>@b = reverse @a</td>
<td>returns reversed version of @a</td>
</tr>
<tr>
<td>shift @a</td>
<td>like pop(@a), but from left-hand end</td>
</tr>
<tr>
<td>unshift @a, x</td>
<td>like push(@a, x), but at left-hand end</td>
</tr>
</tbody>
</table>
Recall the marks example from earlier on; we used "54,67,88" to effectively hold a list of marks.

Could we turn this into a real list if e.g. we wanted to compute an average?

The `split` function allows us to do this:

```bash
split(/pattern/, string)  returns a list
```

The `join` function allows us to convert from list to string:

```bash
join(string, list)  returns a string
```

- Don’t confuse this with the `join` filter in the shell.
  - Perl’s `join` acts more like `paste`
Lists as Strings

Examples:

```
$m = "99,67,85,48,77,84";

@listOfMarks = split(/,/,$marks);
# assigns (99,67,85,48,77,84) to @listOfMarks

$sum = 0;
foreach $m (@listOfMarks) {
    $sum += $m;
}

$newMarks = join(':',$listOfMarks);
# assigns "99:67:85:48:77:84" to $newMarks
```
Lists as Strings

Complex splits can be achieved by using a full regular expression rather than a single delimiter character. If part of the regexp is parenthesised, the corresponding part of each delimiter is retained in the resulting list.

```
# returns (ab,c,d,e)
split(/[#@]+/, 'ab##@#c#d@@e');
# returns (ab,#@#,c,#,d,@@,e)
split(/([#@]+)/, 'ab##@#c#d@@e');
# returns (ab,#,c,#,d,@,e)
split(/([#@])+/, 'ab##@#c#d@@e');
```

And as a specially useful case, the empty regexp is treated as if it matched between every character, splitting the string into a list of single characters:

```
# returns (h, e, l, l, o)
split(//, 'hello');
```
As well as arrays indexed by numbers, Perl supports arrays indexed by strings: **hashes**.

Conceptually, as hash is a set (not list) of \( (key, value) \) pairs.

We can deal with an entire hash at a time via \( \%\text{hashName} \), e.g.

<table>
<thead>
<tr>
<th>#</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
</table>
| %days | "Sun" => "Sunday",
|       | "Mon" => "Monday",
|       | "Tue" => "Tuesday",
|       | "Wed" => "Wednesday",
|       | "Thu" => "Thursday",
|       | "Fri" => "Friday",
|       | "Sat" => "Saturday" );
Individual components of a hash are accessed via `\$hashName\{keyString\}`

Examples:

```perl
$days{"Sun"}  # returns "Sunday"
$days{"Fri"}  # returns "Friday"
$days{"dog"}  # is undefined (interpreted as "")
$days{0}      # is undefined (interpreted as "")
```

# inserts a new (key,value)
```
$days{"dog"} = "Dog Day Afternoon";
```

# replaces value for key "Sun"
```
$days{"Sun"} = "Soonday";
```
Consider the following two assignments:

```
@f = ("John", "blue", "Anne", "red", "Tim", "pink");
    "Tim" => "pink");
```

The first produces an array of strings that can be accessed via position, such as $f[0]$

The second produces a lookup table of names and colours, e.g. $g{"Tim"}$. In fact the symbols => and comma have identical meaning in a list, so either right-hand side could have been used. However, always use the arrow form exclusively for hashes.
Associative Arrays (Hashes)

Consider iterating over each of these data structures:

```perl
foreach $x (@f) {
    print "$x\n";
}
```

```perl
foreach $x (keys %g) {
    print "$x => $g{$x}\n";
}
```

John
blue
Anne
red
Tim
pink

- The data comes out of the hash in arbitrary order.
- This order changes with each execution to make security exploits harder.
- use sort if you need a fixed order
Associative Arrays (Hashes)

There are several ways to examine the \((key, value)\) pairs in a hash:

```perl
foreach $key (keys %myHash) {
    print "($key, $myHash{$key})\n";
}
```

or, if you just want the values without the keys

```perl
foreach $val (values %myHash) {
    print "(?, $val)\n";
}
```

or, if you want them both together

```perl
while (($key,$val) = each %myHash) {
    print "($key, $val)\n";
}
```

- Note that each method produces the keys/values in the same order.
- It’s illegal to change the hash within these loops.
Example (collecting marks for each student):

- a data file of \((\text{name, mark})\) pairs, space-separated, one per line
- out should be \((\text{name, marksList})\), with comma-separated marks

```perl
while (<>){
    chomp;    # remove newline
    ($name, $mark) = split;  # separate data fields
    $marks{$name} .= ',', $mark;  # accumulate marks
}

foreach $name (keys %marks) {
    $marks{$name} =~ s/,//;  # remove comma prefix
    print "$name $marks{$name}\n";
}
```
Deleting An Element from a Hash

The delete function removes an entry (or entries) from an associative array.

# to remove a single pair:
delete $days{"Mon"};  # "I don't like Mondays"

# to remove multiple pairs:
delete @days{ ("Sat","Sun") };  # No weekend!

# to clean out the entire hash:
foreach $d (keys %days) {
  delete $days{$d};
}

# or, more simply
Example - Remembering Snap

```perl
# Reads lines of input until end-of-input
# Print snap! if a line has been seen previously
while (1) {
    print "Enter line: ";
    $line = <STDIN>;
    if (!defined $line) {
        last;
    }
    if ($seen{$line}) {
        print "Snap!\n";
    }
    $seen{$line}++;
}
```

source code for snap_memory.0.pl
# More concise version of snap_memory.0.pl

```perl
while (1) {
    print "Enter line: ";
    $line = <STDIN>;
    last if !defined $line;
    print "Snap!\n" if $seen{$line};
    $seen{$line} = 1;
}
```

source code for snap_memory.1.pl
# Print the nth word on every line of input files/stdin
# output is piped through fmt to make reading easy

die "Usage: $0 <n> <files>\n" if !@ARGV;

$nth_word = shift @ARGV;

open my $f, '|-', "fmt -w 40" or die "Can not run fmt: $!\n";

while ($line = <>) {
    chomp $line;
    @words = split(\ /, $line);
    print $f "$words[$nth_word]\n" if $words[$nth_word];
}

close $f;

---

source code for nth_word.pl
# Perl provides only 1 dimensional arrays
# but arrays elements can contain references to other arrays

```perl
foreach $i (0..3) {
    foreach $j (0..3) {
        $a[$i][$j] = $i * $j;
    }
}

# We can index @a as if it is a 2d-array
# The following loop prints
# 0 0 0 0
# 0 1 2 3
# 0 2 4 6
# 0 3 6 9

foreach $i (0..3) {
    foreach $j (0..3) {
        printf "%2d ", $a[$i][$j];
    }
    print "\n";
}

# @a contains references to 4 arrays
# the following loop will print something like
# ARRAY(0x55ab77d5e120)
# ARRAY(0x55ab77d5e2a0)
# ARRAY(0x55ab77d687c8)
# ARRAY(0x55ab77d68858)

foreach $i (0..3) {
    print "$a[$i]\n";
}

# We can access the whole array referenced by $a[2] as @{$a[2]}
# the following statement prints
# 0 2 4 6
print "@{$a[2]}\n";
```

source code for 2d_array.pl
Example - 2d Array

@a = ();
# assign reference to array to $a[42]
@a[42] = [1,2,3];
print "@{$a[42]}
"; # prints 1 2 3
push @{$a[42]}, (4,5,6);
push @{$a[42]}, (7,8,9);
print "@{$a[42]}
"; # prints 1 2 3 4 5 6 7 8 9

source code for using_2d_array.pl