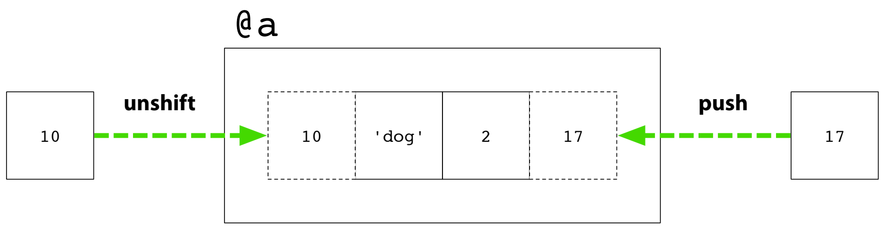
Data structures

Data structures

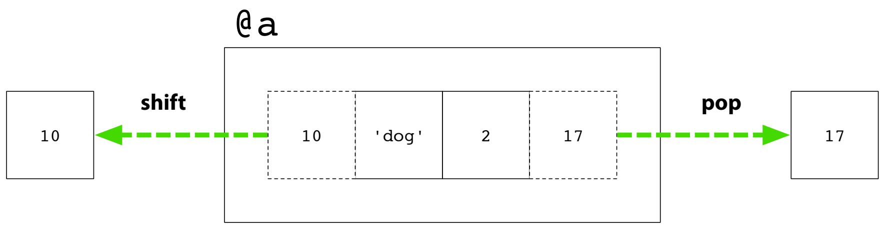
Basic operations
on arrays and hashes

Adding values to an array



```
unshift @a, $val;
push    @a, $val;
```

Removing values from an array



```
$val = shift @a;
$val = pop   @a;
```

Looping (iterating) over the contents of an array

```
foreach my $elem (@a) {
    print "$elem\n";
}
```

$_ – the default iteration variable

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

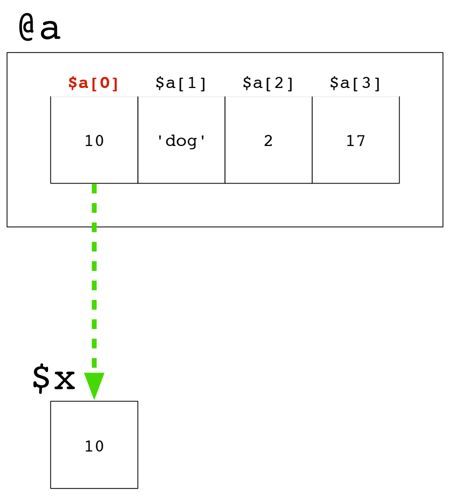
Other ways of iterating over an array

```
while (scalar(@a) != 0) {
    my $elem = shift @a;
    print "$elem\n";
}
```

Shorter – this is a very common idiom!!

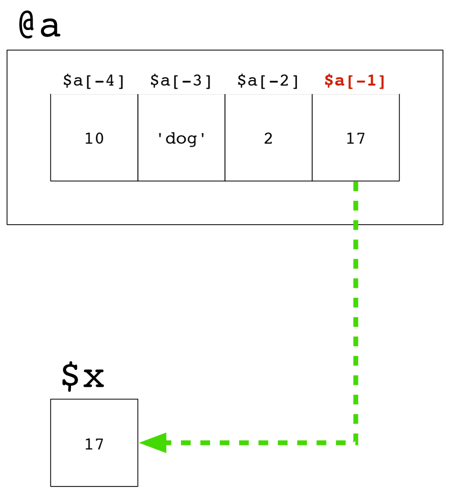
```
while (@a) {
    my $elem = shift @a;
    print "$elem\n";
}
```

Accessing an individual element of an array using its *index*



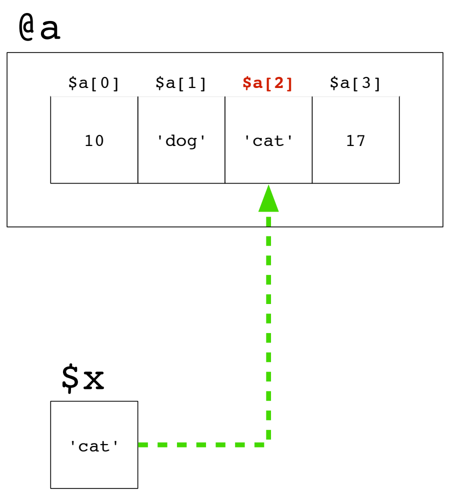
```
$x = $a[0];
```

Negative indexes work from back to front



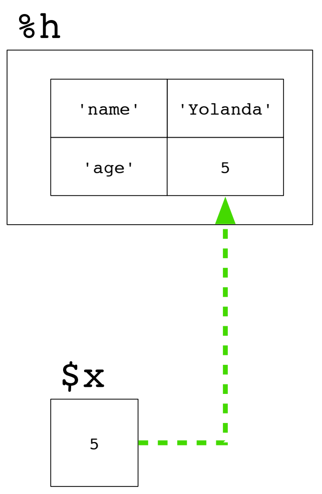
```
$x = $a[-1];
```

Assigning a new value to an existing array element



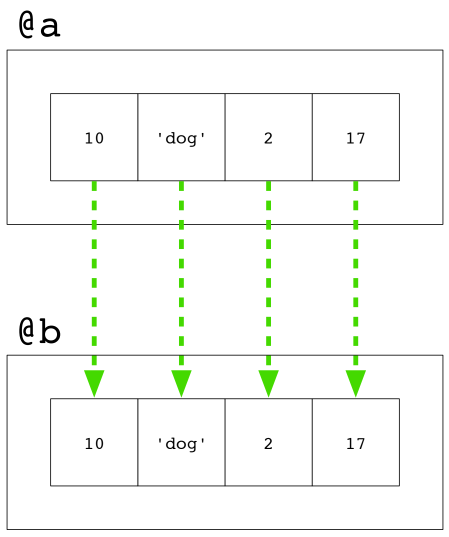
```
$a[2] = $x;
```

Reminder: accessing individual elements in a hash



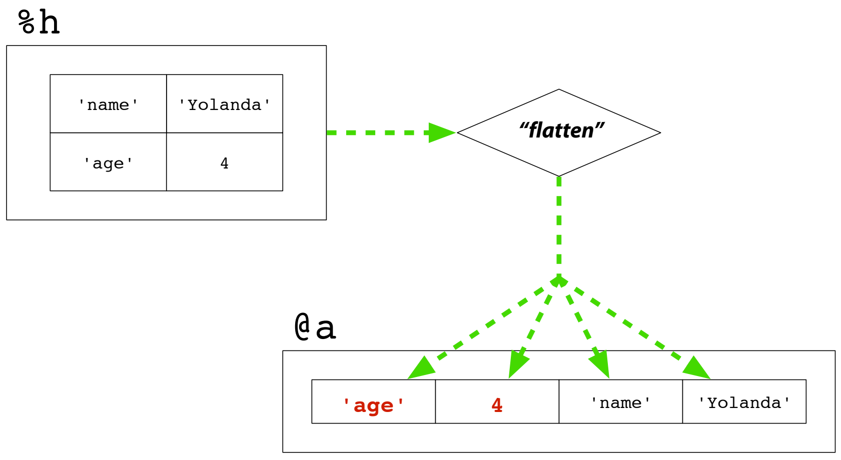
```
$h{'age'} = $x;
```

Copying an array copies the scalar values it contains



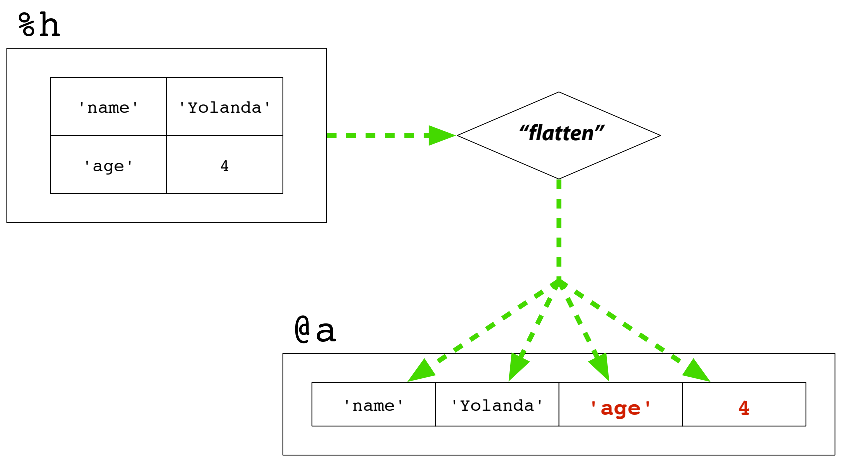
```
@b = @a;
```

Copying a hash into an array



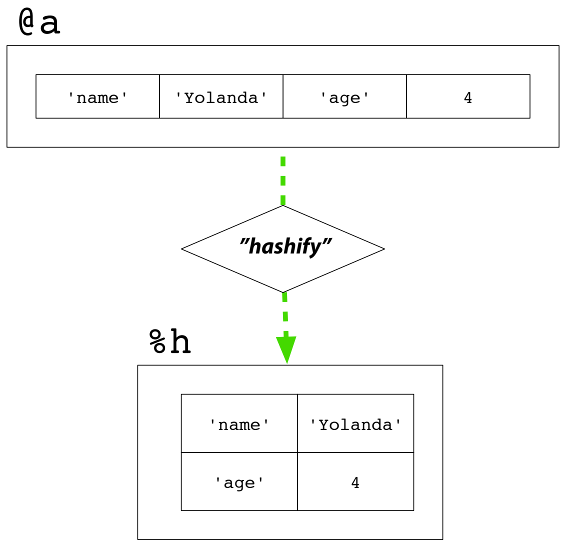
```
@a = %h;
```

Copying a hash into an array (5 milliseconds later)



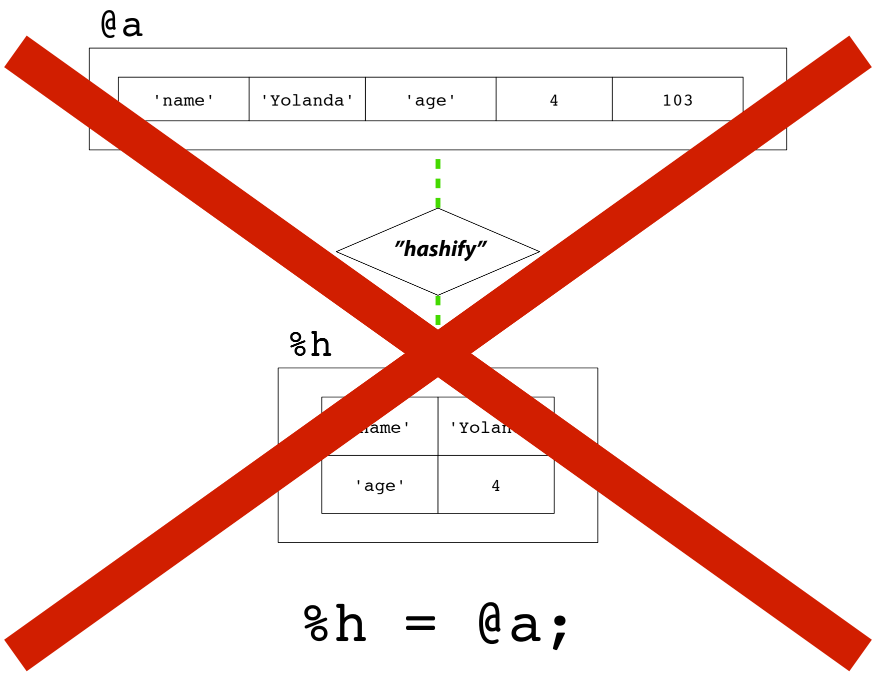
```
@a = %h;
```

Copying an array into a hash



```
%h = @a;
```

The array must have an even number of elements



```
%h = @a;
```

Subroutines

***a.k.a.* functions**

A block

```
{
    print "Hello, world\n";
}
```

The body of a subroutine is a block

```
sub say_hello {
    print "Hello, world\n";
}
```

Calling a subroutine

```
say_hello();
```

Some subroutines always return the same value

```
sub one {
    return 1;
}
```

Some subroutines return many values

```
sub alphabet {
    return ('a', 'b', 'c',
'd', 'e', 'f', 'g', 'h',
'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r',
's', 't', 'u', 'v', 'w',
'x', 'y', 'x');
}
```

Better – use qw()

```
sub alphabet {
    return qw(
      a b c d e f g h i j
      k l m n o p q r s t
      u v w x y z
    );
}
```

Best – use the range operator

```
sub alphabet {
    return ('a'..'z');
}
```

The range operator works with numbers, too

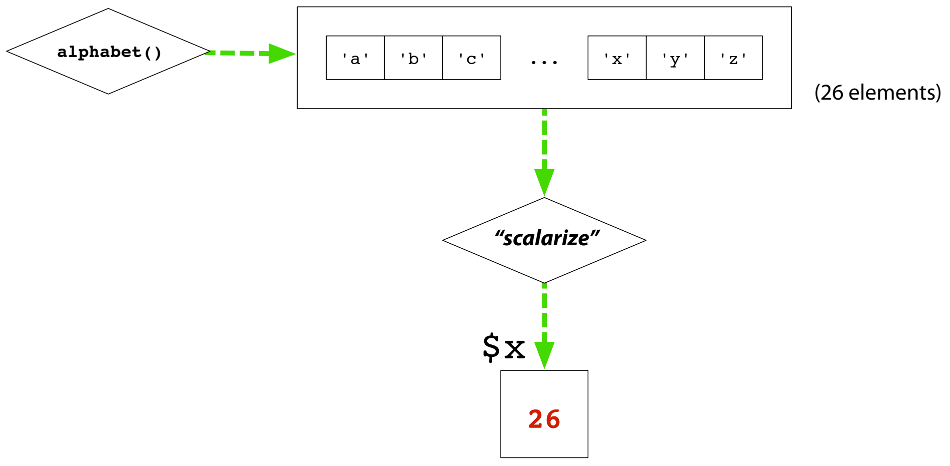
```
sub some_numbers {
    return (-20..30);
}
```

Calling a subroutine in scalar context

```
my $count = alphabet();
```

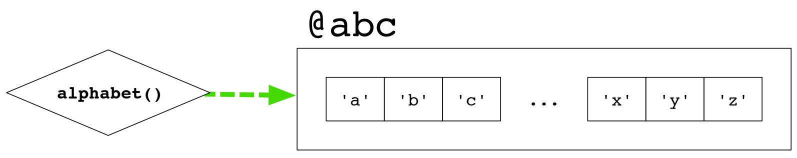
Calling a subroutine in scalar context

```
my $count = alphabet();
```



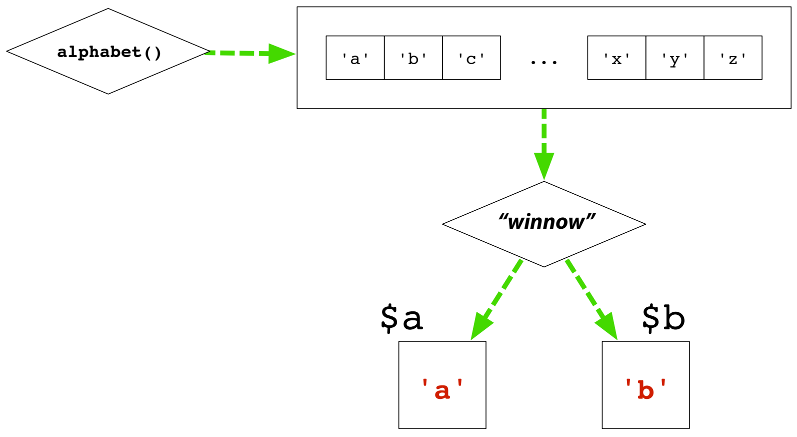
Calling a subroutine in list context

```
my @abc = alphabet();
```



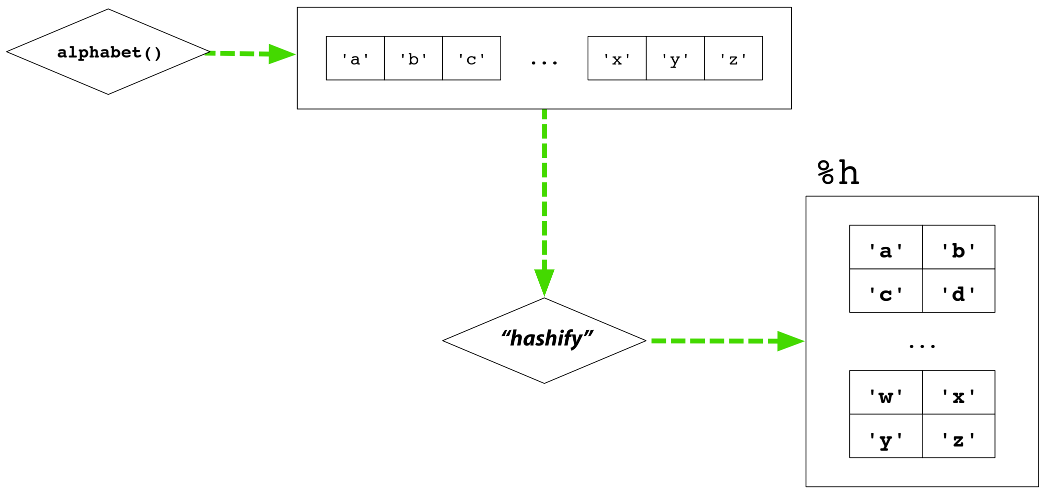
Calling a subroutine in list context

```
my ($a, $b) = alphabet();
```



Calling a subroutine in list context

```
my %h = alphabet();
```



A subroutine that returns different values

```
sub flip_coin {
    my $r = rand();
    if ($r < 0.5) {
        return 'heads';
    }
    else {
        return 'tails';
    }
}
```

Watch the moving hand

```
☞ sub flip_coin {
          my $r = rand();
          if ($r < 0.5) {
               return 'heads';
          }
          else {
               return 'tails';
          }
     }
```

Watch the moving hand

```
     sub flip_coin {
☞         my $r = rand();
          if ($r < 0.5) {
               return 'heads';
          }
          else {
               return 'tails';
          }
     }
```

Value: `0.296903227354539`

Watch the moving hand

```
sub flip_coin {
☞       my $r = rand();
        if ($r < 0.5) {
            return 'heads';
        }
        else {
            return 'tails';
        }
}
```

Value: 0.296903227354539

Watch the moving hand

```
sub flip_coin {
        my $r = rand();
☞       if ($r < 0.5) {
            return 'heads';
        }
        else {
            return 'tails';
        }
}
```

Value: 1

Watch the moving hand

```
    sub flip_coin {
        my $r = rand();
        if ($r < 0.5) {
☞              return 'heads';
        }
        else {
            return 'tails';
        }
    }
```

Value: `'heads'`

Watch the moving hand

```
☞   sub flip_coin {
        my $r = rand();
        if ($r < 0.5) {
            return 'heads';
        }
        else {
            return 'tails';
        }
    }
```

Watch the moving hand

```
sub flip_coin {
☞       my $r = rand();
        if ($r < 0.5) {
            return 'heads';
        }
        else {
            return 'tails';
        }
}
```

Value: `0.922759256881314`

Watch the moving hand

```
sub flip_coin {
☞       my $r = rand();
        if ($r < 0.5) {
            return 'heads';
        }
        else {
            return 'tails';
        }
}
```

Value: `0.922759256881314`

Watch the moving hand

```
sub flip_coin {
    my $r = rand();
☞   if ($r < 0.5) {
        return 'heads';
    }
    else {
        return 'tails';
    }
}
```

Value: `''`

Watch the moving hand

```
sub flip_coin {
    my $r = rand();
    if ($r < 0.5) {
        return 'heads';
    }
    else {
☞       return 'tails';
    }
}
```

Value: `'tails'`

Same thing (read only)

```
sub flip_coin {
    rand() < 0.5
      ? 'heads'
      : 'tails'
      ;
}
```

Most functions take arguments

```
my $name = ask('Name');
```

@_ – gateway into the body of a function

```
my $name = ask('Name');

sub ask {
    my ($label) = @_;
    print "$label: ";
    my $answer = <STDIN>;
    chomp $answer;
    return $answer;
}
```

Multiple arguments – multiple local variables

```
my $sum = add(1, 2);

sub add {
    my ($x, $y) = @_;
    return $x + $y;
}
```

Other ways of handling arguments

```
my $sum = add(1..100);

sub add {
    my $total = 0;
    foreach my $n (@_) {
        $total += $n;
    }
    return $total;
}
```

Other ways of handling arguments

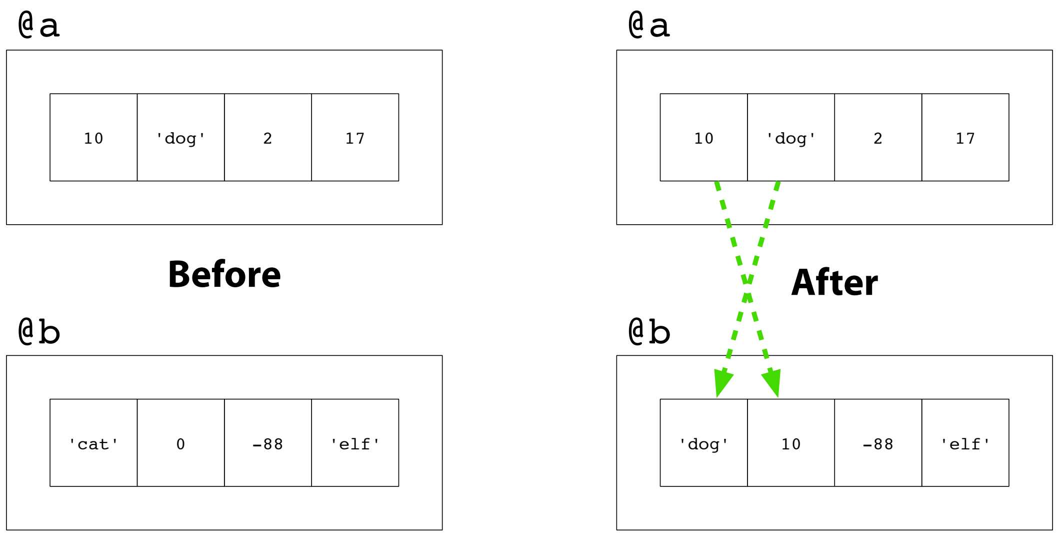
```
my $sum = add(1..100);

sub add {
    my $total = 0;
    while (@_) {
        $total += shift;
    }
    return $total;
}
```

Data structures

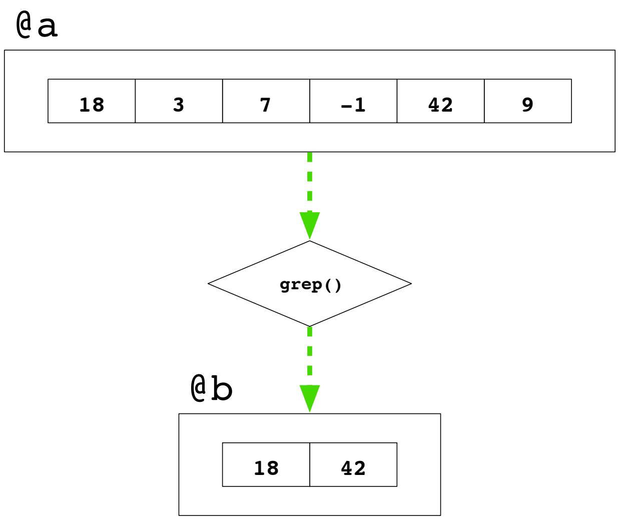
Advanced operations on arrays and hashes

Getting at parts of arrays



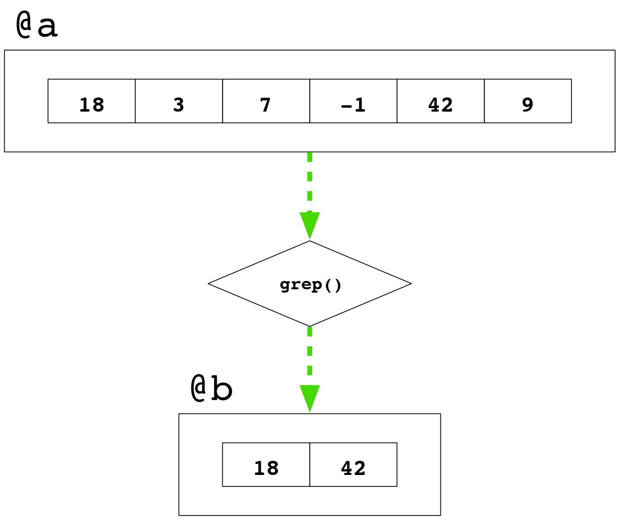
```
@b[0..1] = @a[-3, 0];
```

grep – copy selected elements from an array



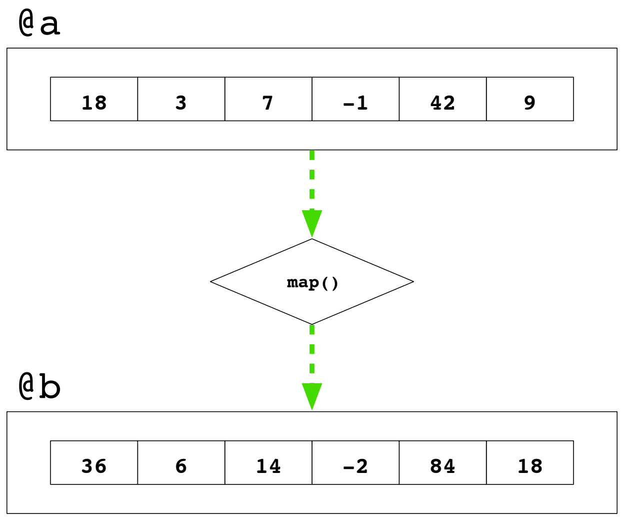
```
@b = grep { $_ > 10 } @a;
```

Alternate syntax – to be avoided



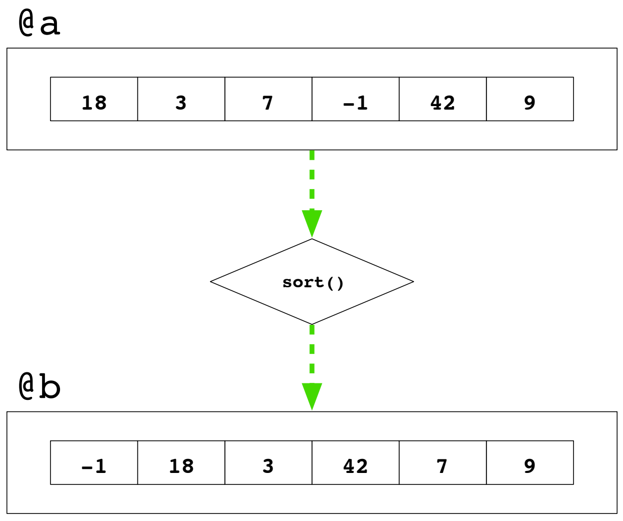
```
@b = grep $_ > 10, @a;
```

map – apply a transformation when copying from an array



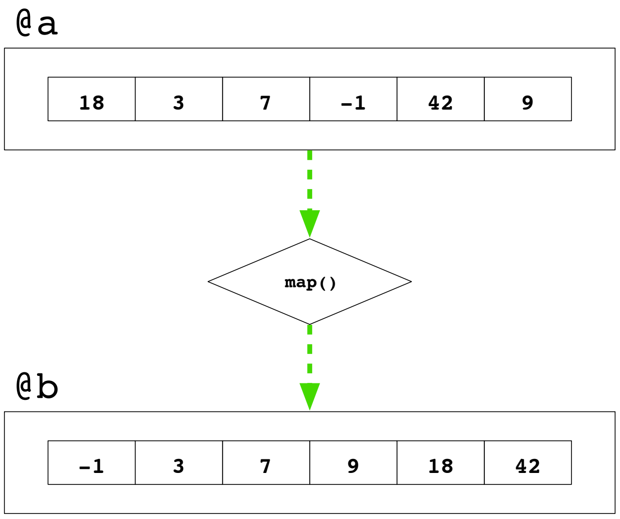
```
@b = map { $_ * 2 } @a;
```

sort – in default (alphabetical) order



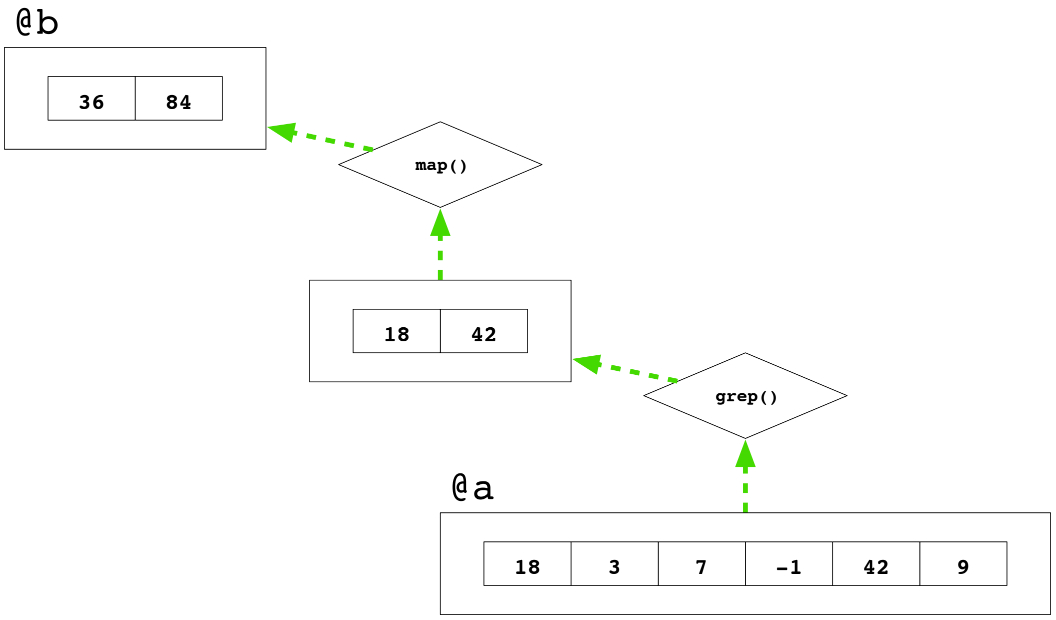
```
@b = sort @a;
```

sort – in numeric order



```
@b = sort { $a <=> $b } @a;
```

Combining map and grep

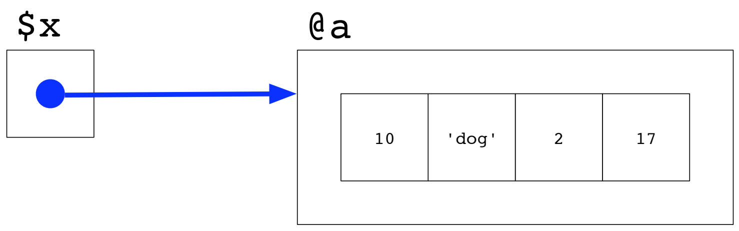


```
@b = map  { $_ * 2  }
     grep { $_ > 10 } @a;
```

Data structures

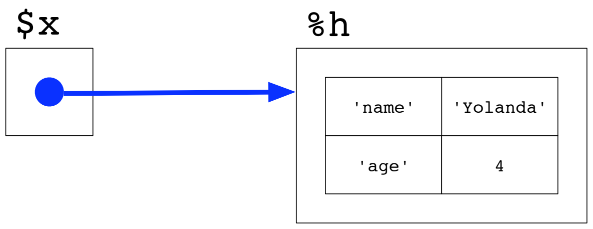
References

A reference to an array is a scalar value



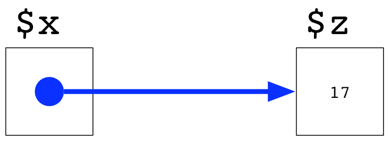
```
$x = \@a;
```

A reference to a hash is a scalar value



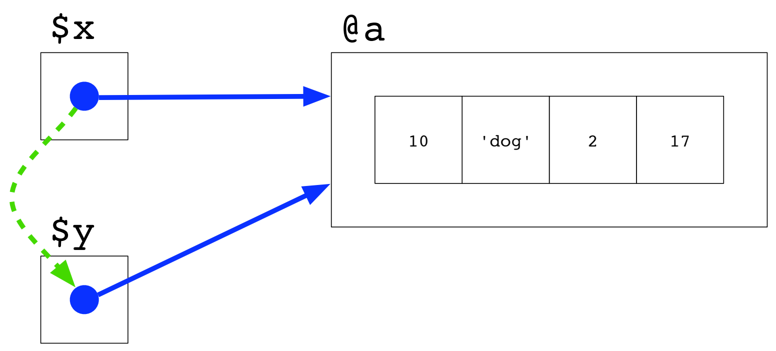
```
$x = \%h;
```

A reference to a scalar is a scalar value



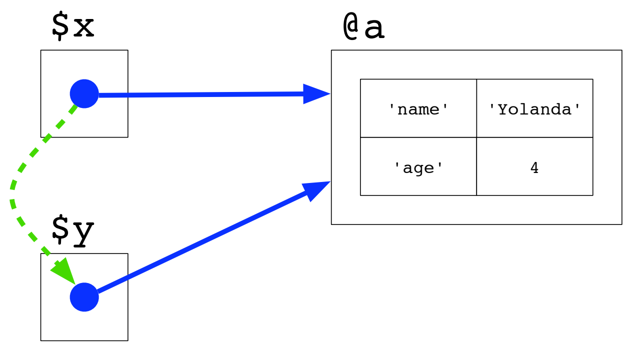
```
$x = \$z;
```

Copying a reference does *not* copy the thing it refers to



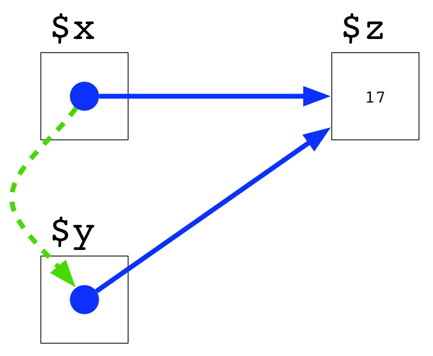
```
$y = $x;
```

Copying a reference does *not* copy the thing it refers to



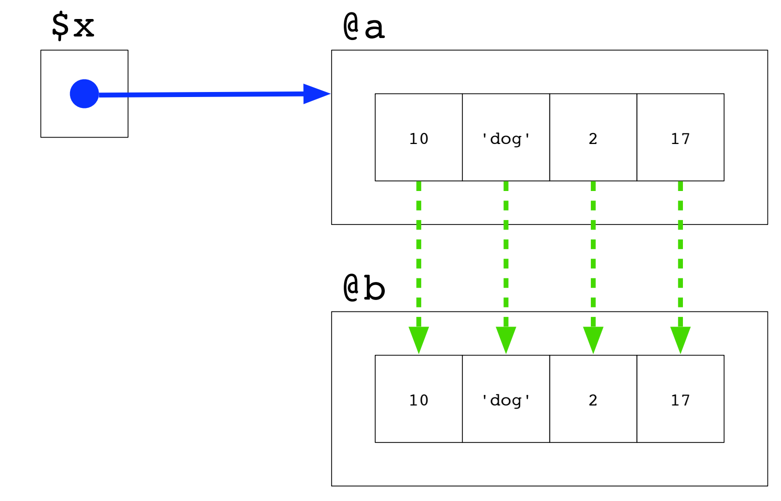
```
$y = $x;
```

Copying a reference does *not* copy the thing it refers to



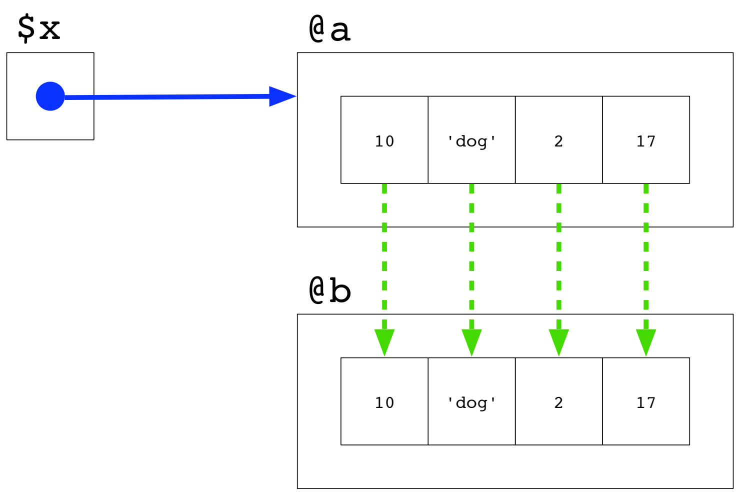
```
$y = $x;
```

Copying an array using a reference to it



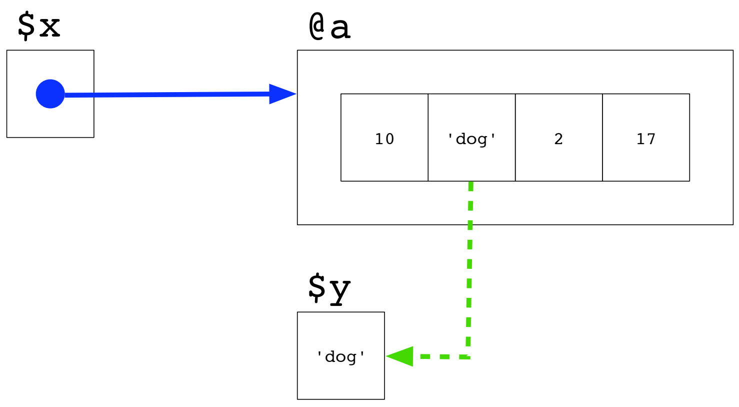
```
@b = @{ $x };
```

Shorthand for the same thing



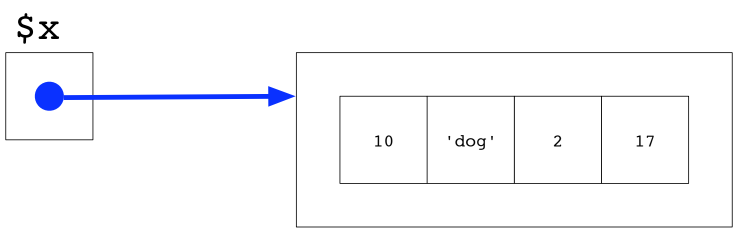
```
@b = @$x;
```

Accessing array elements via a reference to the array



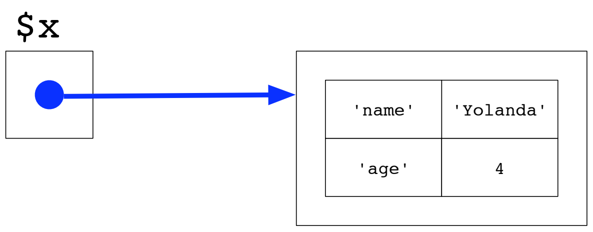
```
$y = $x->[1];
```

A reference to an anonymous array



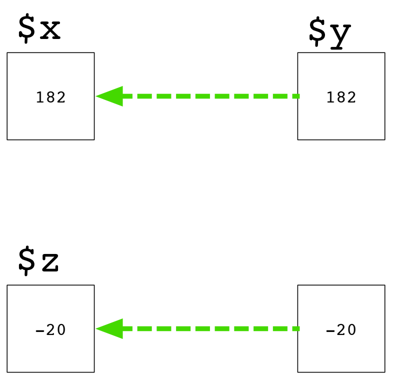
```
$x = [ 10, 'dog', 2, 17 ];
```

A reference to an anonymous hash



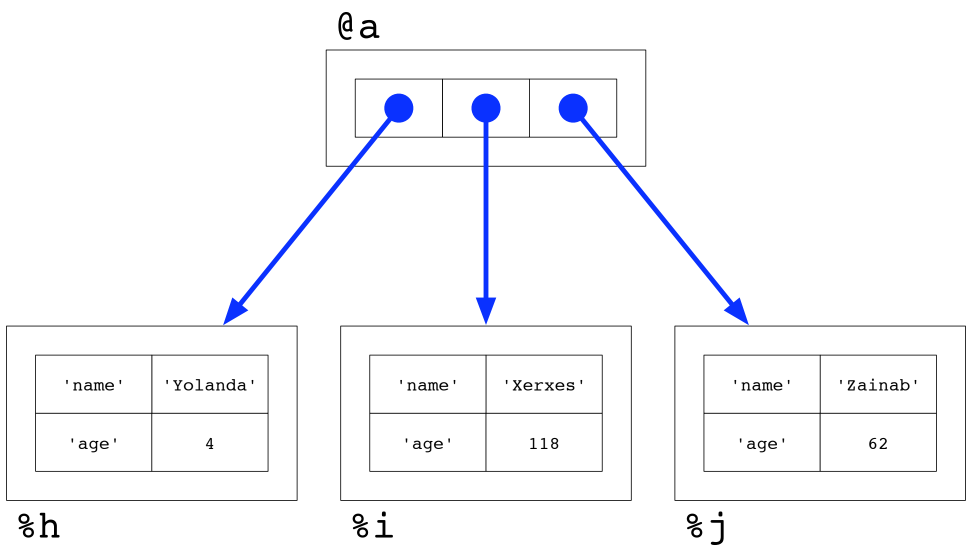
```
$x = { 'name' => 'Yolanda',
       'age'  => 4 };
```

A literal number or string is an anonymous scalar



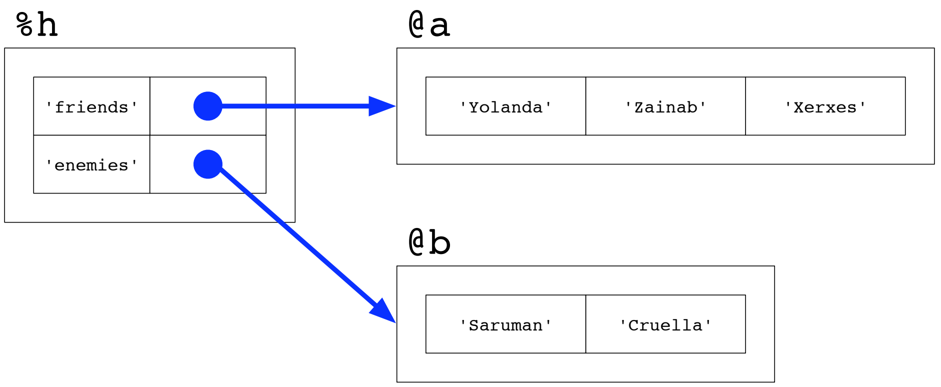
```
$x = $y;
$z = -20;
```

An array of references to hashes



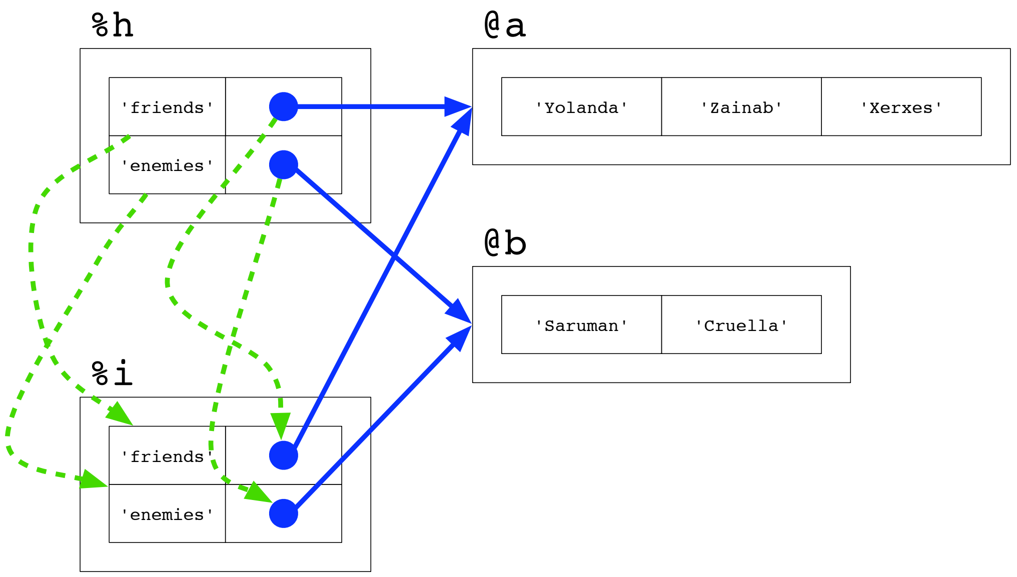
```
@a = ( \%h, \%i, \%j );
```

A hash of references to arrays



```
%h = ( 'friends' => \@a,
       'enemies' => \@b );
```

Copying a hash copies the keys and scalar values it contains



```
%i = %h;
```