

19. Arrays

Perl arrays store lists of scalar values, which may be of different types.

They grow as needed, and may be made to shrink.

Officially, a Perl array is a variable whose value is a list.

A list literal is usually delimited by parentheses.

The parentheses can be omitted if the context allows the list to be identified as such without them.

Examples

```
( ) # the empty list
("cat", 10, "dog", 15) # a list of with different types
($x, $y, $x + $y) # a list with values specified by
# variables and an expression
qw(Fred, Bob) # a list whose elements are implicitly
# quoted
```

The range operator, `..` (two dots), specifies a range of scalar literals – e.g.,

```
(1 .. 4) # the list (1, 2, 3, 4)
(1, 3 .. 6, 8) # the list (1, 3, 4, 5, 6, 8)
(3 .. 1) # the list ()
('a' .. 'd') # the list ('a', 'b', 'c', 'd')
```

The name of an array variable begins with a @.

Array variables and scalar variables are in different name spaces.

An uninitialized array has () as its value.

If an assignment has an array variable as its left side (target), then

- an array is assigned as a unit no matter what the form of the right side.

If the right side is an array or list literal with more elements than the target array, then

- the target array is extended to accommodate it.

If there are fewer elements on the right side, then

- the excess elements in the target are set to undef.

If the right side is a scalar value, then

- the target will be an array of length one, containing just that value.

If an array is assigned to a scalar variable, then

- the array's length is assigned to the target.

Examples

```
@ar = "cat";      # @ar is ("cat")
@ar = (1, 3, 5);  # @ar now has 3 elements
@ar = (4);       # @ar still has 3 elements, but the last 2
                 # are undef
$len = @ar;      # $len is 3
```

A list of variables may be the target of an array assignment.

Examples

```
($a, $b) = ("cat", "dog");
           # $a is "cat", $b is "dog"
($a, $b) = ($b, $a);
           # Now $a is "dog", $b is "cat"
```

If there are fewer variables in the target list than elements in the right side, then

- the excess elements are ignored.

If there are more variables than right-side elements, then

- the excess variables are set to undef.

Examples

```
($a, $b) = ("fish", "cat", "dog");
           # $a is "fish", $b is "cat"
($a, $b) = ("bird");
           # $a is "bird", $b is undef
```

If an array variable appears in the target list, all the rest of the right-side elements are assigned to it as a list.

For example,

```
($a, @ar, $b) = (2, 4, 6);
           # $a is 2, @ar is (4, 6), $b is undef
($first, @ar) = @ar;
           # $first is 4, @ar is now (6)
```

An array in a list literal that isn't the target of an assignment is expanded to a list of its values.

For example, if `@ar` is `(2 , 3)`, then the value of

```
( 1 , @ar , 4 )
```

is

```
( 1 , 2 , 3 , 4 )
```

The `undef` operator sets its array argument to `()`.

```
undef @ar;
```

When an array variable is interpolated in a double-quoted string, the values of the array elements are converted to strings and concatenated together separated by spaces.

For example,

```
@ar = ( 1 , 2 , 3 );
print "The array is: @ar\n";
```

outputs

```
The array is: 1 2 3
```

If we replace the `print` statement here with

```
print @ar;
```

we get

```
123
```

The first index in an array is 0.

We access the element at index `$i` in array `@ar` with

```
$ar[ $i ]
```

If there is also a scalar variable `$ar`, there is no connection between it and `$ar[$i]`.

An access to an element beyond the last returns `undef`.

An array element name can appear in a list literal – e.g.,

```
($ar[0], $ar[1]) = ($ar[1], $ar[0])
```

A negative subscript is an offset from the right end of the array.

Subscript `-1` indexes the last element.

For example, given

```
@ar = ( 2, 4, 6 );
```

then

```
$ar[ -1 ] is 6
```

```
$ar[ -2 ] is 4
```

```
$ar[ -3 ] is 2
```

Assigning a value to an element beyond the last element in an array causes the array to be extended to the required length.

For example:

```
@ar = ( 2 );
```

```
$ar[ 2 ] = 6;
```

```
# $ar[0] is 2, $ar[1] is undef, $ar[2] is 6
```

The index of the last element of array `@ar` is the value of the implicit variable `$#ar`.

We can set `@ar` to `()` with

```
 $#ar = -1;
```

If we know that the maximum size of `@ar` will be say, 100, it is best to make it this big at the start.

Either of the following will do:

```
 $ar[ 99 ] = undef;
 $#ar = 99;
```

Example

The following program first reads a file of sorted names (one per line) whose name is given as a command-line argument.

It builds an array of the names as it reads them in.

It then inserts names input from the keyboard (one per line) until the user presses Control-Z (Control-D in UNIX).

It inserts these names in the proper order in the sorted array.

The new name is compared with the names in the array, starting with the last.

Each name that belongs after the new name is moved down one position.

When the correct position is found, the new name is inserted.

Note that the array grows whenever a new name is added.

```

# Get the first list of names from the file into
# an array

$index = 0;

while ( chomp( $names[ $index ] = <> ) ) {
    $index++;
}

# Loop to read and insert the names from the
# second list. Move names down, starting at the
# bottom, until the position of the new name
# is found.

while ( chomp( $new_name = <STDIN> ) ) {
    for ( $index = $#names;
          $index >= 0
          and $names[$index] gt $new_name;
          $index-- ) {
        $names[ $index + 1 ] = $names[ $index ];
    }

    # Insert the new name:
    $names[ $index + 1 ] = $new_name;
}

print "\nThe complete list:\n\n";

for ( $index = 0; $index <= $#names; $index++ ) {
    print "$names[ $index ] \n";
}

```

Slices

In Perl, a slice of an array is a reference to some subset of the elements of the array in some specified order.

A slice is specified by

- the array name and,
- within `[...]`, a sequence of subscripts in the array's range.

Because a slice is an array, its name begins with a `@`.

The subscripts can be specified individually or using a range operator.

They can include expressions.

A slice can be the target of an assignment.

For example, given

```
@ar = (2, 4, 6, 8, 10);
```

then

```
@ar[ 2 .. 4 ]      is (6, 8, 10)
@ar[ 0, 2 .. 4 ]  is (2, 6, 8, 10)
@ar[ 0, 2, 4 ]    is (2, 6, 10)
@ar[ 4, 2, 0 ]    is (10, 6, 2)
@ar[ 0 .. $#ar - 2 ] is (2, 4, 6)
```

And, if we execute

```
@ar[ 0, $#ar ] = @ar[ $#ar, 0 ];
then @ar is (10, 4, 6, 8, 2).
```

Scalar and List Contexts

Perl determines the value of an operand partly by the context of the operand.

The two primary contexts are scalar and list (or array).

In an assignment, the target determines the context hence the form of the right side value.

So, if the target is a scalar and the right side is an array, then the array is coerced to a scalar, namely, the size of the array.

And, if the target is an array, the context of the right side is list.

So, if the right side is a scalar, it's coerced to a list of one element.

Some functions and operators require their parameters/operands to be scalars, others require lists.

We'll indicate which when we present these functions and operators.

The context can be forced to be scalar by making an array the operand of a scalar operator.

For example,

```
@ar + 0
```

evaluates to the length of @ar.

Scalar context can be explicitly forced with the `scalar` function.

For example, consider

```
print "Length: ", scalar( @ar ), "\n";
```

`print` takes a list of strings as its operand.

This list form is needed to keep the call to `scalar` outside the string literal.

Function calls are not interpolated.

A list literal can't be forced to scalar context.

Some operators work in both scalar and list contexts.

For example, when the line input operator appears in scalar context, as in

```
$next = <STDIN>
```

it returns the next line of keyboard input.

When it appears in list context, as in

```
@names = <STDIN>
```

all lines of keyboard input are read and assigned to the `@names` array.

The `foreach` Statement

The general form of a `foreach` statement is

```
foreach scalar_variable ( list_or_array ) {
  ...
}
```

The scalar variable is local to the loop body.

It becomes an alias for the elements of the list or array, one after another, for successive iterations.

Examples

```
$sum = 0;
foreach $val ( 2, 4, 6 ) {
  $sum += $val;
}
```

results in `$sum` being 12.

```
foreach $num ( @numbers ) {
  $num++;
}
```

increments all the elements of `@numbers` – `$num` is an alias for successive elements of `@numbers`.

```
$sum = 0;
foreach $index ( 0 .. 9 ) {
  $sum += $numbers[ $index ];
}
```

results in `$sum` being the sum of the first ten elements of `@numbers`.

If the control variable is omitted, `$_` is used in its place.

Example

```
foreach ( 0 .. 2 ) {  
    print;  
}
```

outputs 012.

`foreach` as a statement modifier is used in the form

statement `foreach` *expression*,

where *expression* is a list literal or array and does not include a control variable.

Example

```
$sum = 0;  
$sum += $numbers[ $_ ] foreach ( 0 .. 9 );
```

List Operators

`reverse` returns a list in which the order of the elements of its operand list are reversed.

It does not change the list to which it's applied. (No side effect.)

For example,

```
@nums = (1, 2, 3);
@rnums = reverse @nums;
sets @rnums to (3, 2, 1).
```

`sort`, applied to a list of strings, returns the list in alphabetical order with no side effect.

For example,

```
@names = sort ( "Fred", "Bill", "Ed" );
sets @names to ("Bill", "Ed", "Fred").
```

Applied to a list of numbers, it coerces them to strings and sorts the resulting list in alphabetical order.

Other orders can be specified, but this requires a function operand and will be discussed later.

`x`, the repetition operator introduced earlier for strings, also works on lists.

For example,

```
@nums = (1, 2) x 2;
sets @nums to (1, 2, 1, 2).
```

This lets you set all elements of an array to a given value – e.g.,

```
@ar = (0) x @ar;
```

Here the value of `@ar` on the right side is the length of `@ar` since the right operand or `x` is a scalar.

When given a list operand,

- `chomp` removes the input record separator from all elements and returns the number removed, and
- `chop` removes the last character of each element and returns the character removed from the first element.

`push` takes an array name and a scalar, pushing the scalar onto the end of the array (the top of the stack).

`pop` takes an array name and pops the value at the end of the array.

It returns this value or, if the array is empty, `undef`.

`shift` and `unshift` do the same thing as `pop` and `push`, respectively, but at the beginning of the array.

Example

The following program reads a sequence of first names from a file whose name is supplied as a command-line argument.

There is one name per line.

For each first name read, it prompts for and inputs the last name.

The entire name is then pushed onto an array called `@names`.

Next, for each name in `@names`, the program prompts for and reads the age of the person.

A string with the name and age is pushed onto an array called `@info`.

Finally, all the information is printed to the screen, one line per person.

If the user fails to supply information for a prompt (presses Control-Z), the program identifies what is missing and terminates.

```

# arrays2.pl

while ( chomp( $first_name = <> ) ) {
    print "The last name of $first_name: ";
    chomp( $last_name = <STDIN> )
        or die "No last name given for $first_name";
    push @names, "$first_name $last_name";
}

foreach (@names) {
    print "$_\n's age: ";
    chomp( $age = <STDIN> )
        or die "No age given for $_";
    push @info, "$_, $age years old\n";
}

print "\n\nThe people are:\n@info";

```

Neither of the two arrays are explicitly initialized – they are initially ().

The second, `foreach` loop uses the implicit control variable `$_`.

The following are the contents of the data file, `arrays2.dat`:

```
Bob
Fred
Al
```

The following is an example run using this data file:

```
C:\someDirectory>perl arrays2.pl arrays2.dat
The last name of Bob: Smith
The last name of Fred: Jones
The last name of Al: Green
Bob Smith's age: 21
Fred Jones's age: 35
Al Green's age: 17

The people are:
Bob Smith, 21 years old
Fred Jones, 35 year old
All Green, 17 years old
```

The following is a run where the user fails to supply a last name.

```
C:\someDirectory>perl arrays2.pl arrays2.dat
The last name of Bob: Smith
The last name of Fred: ^Z
No last name given for Fred at arrays2.pl line 5, <STDIN> line 1.
```

`splice` is used primarily for its side effect.

Given an array and an index into the array,

- `splice` modifies the array in place by removing all elements from that index to the end of the array.

It returns the slice of elements removed.

For example,

```
@ar = ( 2, 4, 6, 8 );
@ar1 = splice( @ar, 2 );
results in @ar being (2, 4) and @ar1 being (6, 8).
```

If a third argument is supplied, it specifies the number of elements to remove.

For example,

```
@ar = ( 2, 4, 6, 8 );
splice( @ar, 1, 2 );
results in @ar being (2, 8).
```

We can supply an array as a fourth argument.

Its elements replace those removed from the array.

For example,

```
@ar = ( 2, 4, 6, 8 );
@new = ( 3, 5 );
splice( @ar, 1, 2, @new );
results in @ar being (2, 3, 5, 8).
```

In place of a fourth, array argument, we can include any number of scalar values after the third argument.

Then `splice` behaves as if an array of these scalars were used as the fourth argument.

For example,

```
@ar = (2, 4, 6, 8);  
splice( @ar, 1, 2, 3, 5);  
results in @ar being (2, 3, 5, 8).
```

`splice` can be used to insert elements (without removing any) by giving it 0 as the third argument.

For example,

```
@ar = (2, 4, 10);  
splice( @ar, 1, 0, 6, 8);  
results in @ar being (2, 4, 6, 8, 10).
```

`split` splits a string into an array of substrings.

It's used in the form

```
split / pattern / , string , limit
```

where

- *pattern* is a regular expression specifying where to split *string*, and
- *limit* is the maximum number of substrings that `split` should produce.

All the substrings produced are returned as a list.

For example,

```
@names = split /, */, "Bob, Ed,Al, Pat", 3;
sets @names to
("Bob", "Ed", "Al, Pat").
```

When *pattern* matches nothing in *string*, `split` returns a list containing only *string*.

If *limit* is missing, *string* is split into as many substrings as possible.

For example,

```
@names = split /, */, "Bob, Ed,Al, Pat";
sets @names to
("Bob", "Ed", "Al", "Pat").
```

If *string* is also missing, the implicit variable `$_` is used.

If *pattern* is missing as well, any amount of whitespace is used as the separator.

For example,

```
$_ = "a      b c d";
@letters = split;
sets @letters to ("a", "b", "c", "d").
```

If *pattern* is the empty expression, `split` splits *string* into its individual characters.

For example,

```
@letters = split //, "cat";
sets @letters to ("c", "a", "t").
```

If there is an occurrence of *pattern* at the end of *string*, `split` ignores it.

`split` doesn't then include an empty string in the result.

But, `split` produces an empty string for a pair of adjacent occurrences of *pattern*.

For example,

```
@ar = split /:/, "a:b::c:";
sets @ar to ("a", "b", "", "c").
```

Example

The following reads a line at a time from the file whose name is given as a command-line argument.

Each line of this file contains a person's first and last names.

The program splits each name, storing the first name in @first_names and the last name in @last_names.

Note that the input record separator counts as whitespace here.

Finally, the program outputs the first names then the last names.

```
# split.pl
while ( <> ) {
    ( $first_names[ $i ], $last_names[ $i++ ] )
    = split;
}

print "The first names are: @first_names\n";
print "The last names are: @last_names\n";
```

The following are the contents of the data file, split.dat:

```
Bob Smith
Fred Jones
Al Smith
```

The following is an example run using this data file:

```
C:\someDirectory>perl split.pl split.dat
The first names are: Bob Fred Al
The last names are: Smith Jones Smith
```