Chapter 3. Lists and Arrays

If a scalar is the “singular” in Perl, as we described it at the beginning of Chapter 2, the “plural” in Perl is represented by lists and arrays.

A *list* is an ordered collection of scalars. An *array* is a variable that contains a list. People tend to use the terms interchangeably, but there’s a big difference. The list is the data and the array is the variable that stores that data. You can have a list value that isn’t in an array, but every array variable holds a list (although that list may be empty). Figure 3-1 represents a list, whether it’s stored in an array or not.

Since lists and arrays share many of the same operations, just like scalar values and variables do, we’ll treat them in parallel. Don’t forget their differences though.

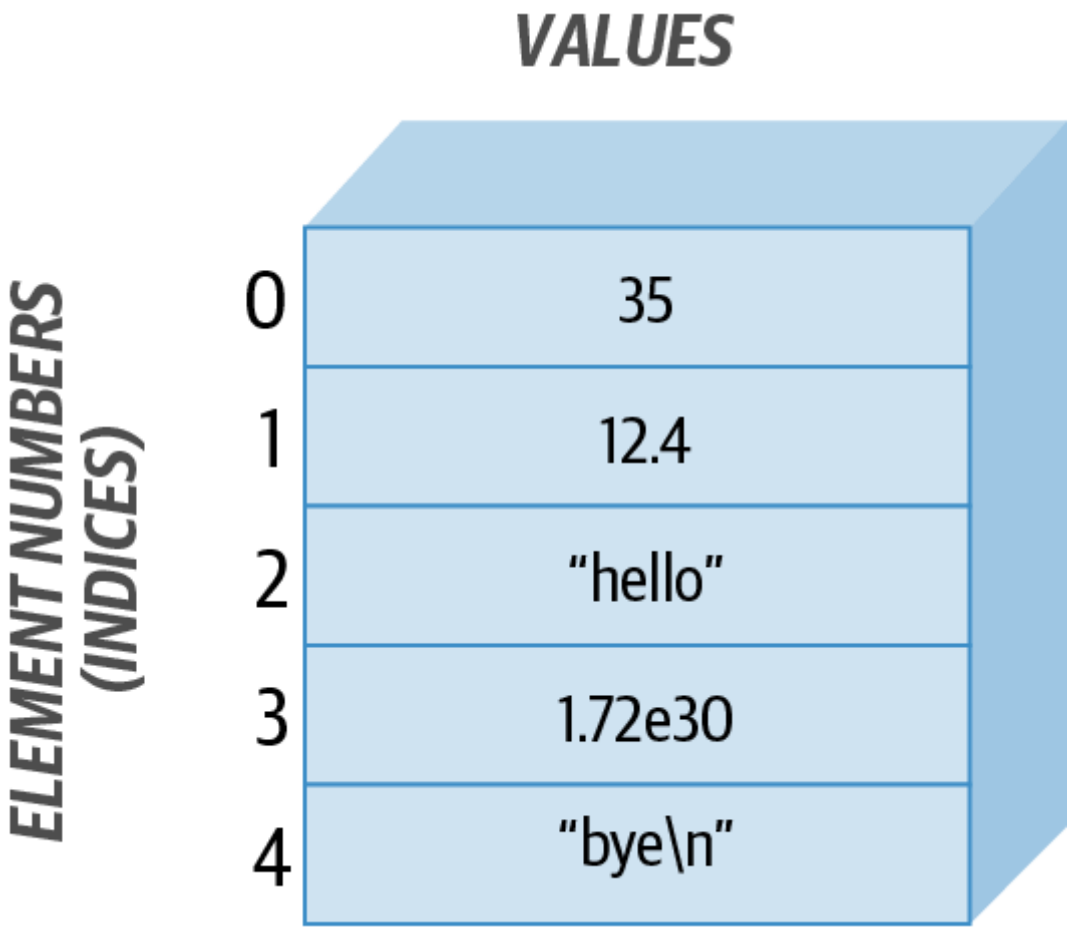

Figure 3-1. A list with five elements

Each *element* of an array or list is a separate scalar value. These values are ordered—that is, they have a particular sequence from the first to the last element. The elements of an array or a list are *indexed* by integers starting at zero and counting by ones, so the first element of any array or list is always element zero. This also means that the last index is one less than the number of items in the list.

Since each element is an independent scalar value, a list or array may hold numbers, strings, `undef` values, or any mixture of different scalar values. Nevertheless, it's common to have all elements of the same type, such as a list of book titles (all strings) or a list of cosines (all numbers).

Arrays and lists can have any number of elements. The smallest one has no elements, while the largest can fill all of available memory. Once again, this is in keeping with Perl's philosophy of "no unnecessary limits."

Accessing Elements of an Array

If you've used arrays in another language, you won't be surprised to find that Perl provides a way to *subscript* an array in order to refer to an element by a numeric index.

The array elements are numbered using sequential integers, beginning at 0 and increasing by 1 for each element, like this:

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

The array name itself (in this case, `fred`) is from a completely separate namespace than scalars use; you can have a scalar variable named `$fred` in the same program, and Perl will treat them as different things and won't be confused. (Your maintenance programmer might be confused, though, so don't capriciously make all of your variable names the same!)

You can use an array element like `$fred[2]` in (almost) every place where you could use any other scalar variable. For example, you can get the value from an array element or change that value by the same sort of expressions you used in Chapter 2:

```
print $fred[0];
$fred[2]  = "diddley";
$fred[1] .= "whatsis";
```

The subscript may be any expression that gives a numeric value. If it's not an integer already, Perl will automatically truncate it (not round!) to the whole number portion:

```
$number = 2.71828;
print $fred[$number - 1]; # Same as printing $fred[1]
```

If the subscript indicates an element that would be beyond the end of the array, the corresponding value will be `undef`. This is just as with ordinary scalars; if you've never stored a value in the variable, it's `undef`:

```
$blank = $fred[ 142_857 ]; # unused array element gives undef
$blanc = $mel;             # unused scalar $mel also gives undef
```

Special Array Indices

If you assign to an array element that is beyond the end of the array, the array is automatically extended as needed—there's no limit on its length, as long as there's available memory for Perl to use. If Perl needs to create the intervening elements, it creates them as `undef` values:

```
$rocks[0]  = 'bedrock';      # One element...
$rocks[1]  = 'slate';        # another...
$rocks[2]  = 'lava';         # and another...
$rocks[3]  = 'crushed rock'; # and another...
$rocks[99] = 'schist';       # now there are 95 undef elements
```

Sometimes you need to find out the last element index in an array. For the array of `rocks`, the last element index is `$#rocks`. That's not the same as the number of elements, though, because there's an element number zero:

```
$end = $#rocks;                  # 99, which is the last element's index
$number_of_rocks = $end + 1;     # OK, but you'll see a better way later
$rocks[ $#rocks ] = 'hard rock'; # the last rock
```

Using the `$#name` value as an index, like that last example, happens often enough that Larry has provided a shortcut: negative array indices count from the end of the array. But don't get the idea that these indices "wrap around." If you have three elements in the array, the valid negative indices are `-1` (the last element), `-2` (the middle element), and `-3` (the first element). If you try `-4` and beyond, you just get `undef`. In the real world, nobody seems to use any of these except `-1`, though:

```
$rocks[ -1 ]   = 'hard rock';   # easier way to do that last example
$dead_rock     = $rocks[-100];  # gets 'bedrock'
$rocks[ -200 ] = 'crystal';     # fatal error!
```

List Literals

A *list literal* (the way you represent a list value within your program) is a list of comma-separated values enclosed in parentheses. These values form the elements of the list. For example:

```
(1, 2, 3)      # list of three values 1, 2, and 3
(1, 2, 3,)     # the same three values (the trailing comma is ignored)
("fred", 4.5)  # two values, "fred" and 4.5
( )            # empty list - zero elements
```

You don't have to type out every value if they are in sequence. The `..` *range operator* creates a list of values by counting from the left scalar up to the right scalar by ones. For example:

```
(1..100)          # list of 100 integers
(1..5)            # same as (1, 2, 3, 4, 5)
(1.7..5.7)        # same thing; both values are truncated
(0, 2..6, 10, 12) # same as (0, 2, 3, 4, 5, 6, 10, 12)
```

The range operator only counts up, so this won't work and you'll get the empty list:

```
(5..1)            # empty list; .. only counts "uphill"
```

The elements of a list literal are not necessarily constants; they can be expressions that will be newly evaluated each time the literal is used. For example:

```
($m, 17)       # two values: the current value of $m, and 17
($m+$o, $p+$q) # two values
($m..$n)       # range determined by current values of $m and $n
(0..$#rocks)   # the indices of the rocks array from the previous section
```

The qw Shortcut

A list may have any scalar values, like this typical list of strings:

```
("fred", "barney", "betty", "wilma", "dino")
```

It turns out that lists of simple words (like the previous example) are frequently needed in Perl programs. The `qw` shortcut makes it easy to generate them without typing a lot of extra quote marks:

```
qw( fred barney betty wilma dino ) # same as earlier, but less typing
```

`qw` stands for "quoted words" or "quoted by whitespace," depending on whom you ask. Either way, Perl treats it like a single-quoted string (so, you can't use `\n` or `$fred` inside a `qw` list as you would in a double-quoted string). The whitespace (characters like spaces, tabs, and new-

lines) disappear and whatever is left becomes the list of items. Since whitespace is insignificant, here's another (but unusual) way to write that same list:

```
qw(fred
  barney     betty
wilma dino)  # same as before, but pretty strange whitespace
```

Since `qw` is a form of quoting, though, you can't put comments inside a `qw` list. Some people like to format their lists with one element per line, which makes it easy to read as a column:

```
qw(
    fred
    barney
    betty
    wilma
    dino
)
```

The previous two examples used parentheses, but Perl lets you choose any punctuation character as the delimiter. Here are some of the common ones:

```
qw! fred barney betty wilma dino !
qw/ fred barney betty wilma dino /
qw# fred barney betty wilma dino #   # like in a comment!
```

Sometimes the two delimiters can be different. If the opening delimiter is one of those "left" characters, the corresponding "right" character is the proper closing delimiter:

```
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
```

If you need to include the closing delimiter *within* the string as one of the characters, you probably picked the wrong delimiter. But even if you can't or don't want to change the delimiter, you can still include the character using the backslash:

```
qw! yahoo\! google ask msn ! # include yahoo! as an element
```

As in single-quoted strings, two consecutive backslashes contribute one single backslash to the item:

```
qw( This as a \\ real backslash );
```

Now, although the Perl motto is "There's More Than One Way To Do It," you may wonder why anyone would need all of those different ways! Well, you'll see later that there are other kinds of quoting where Perl uses this same rule, and it can come in handy in many of those. But even here, it could be useful if you need a list of Unix filenames:

```
qw{
    /usr/dict/words
    /home/rootbeer/.ispell_english
}
```

That list would be quite inconvenient to read, write, and maintain if you could only use the `/` as a delimiter.

List Assignment

In much the same way as you can assign scalar values to variables, you can assign list values to variables:

```
($fred, $barney, $dino) = ("flintstone", "rubble", undef);
```

All three variables in the list on the left get new values, just as if you did three separate assignments. Since the list on the right side is built up be-

fore the assignment starts, this makes it easy to swap two variables' values in Perl:

```
($fred, $barney) = ($barney, $fred); # swap those values
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

But what happens if the number of variables (on the left side of the equals sign) isn't the same as the number of values (from the right side)? In a list assignment, extra values are silently ignored—Perl figures that if you wanted those values stored somewhere, you would have told it where to store them. Alternatively, if you have too many variables, the extras get the value `undef` (or the empty list, as you'll see in a moment):

```
($fred, $barney) = qw< flintstone rubble slate granite >; # two ignored items
($wilma, $dino)  = qw[flintstone];                        # $dino gets undef
```

Now that you can assign lists, you *could* build up an array of strings with a line of code like this:

```
($rocks[0], $rocks[1], $rocks[2], $rocks[3]) = qw/talc mica feldspar quartz/;
```

But when you wish to refer to an entire array, Perl has a simpler notation. Use the at sign (`@`) before the name of the array (and no index brackets after it) to refer to the entire array at once. You can read this as "all of the," so `@rocks` is "all of the rocks." This works on either side of the assignment operator:

```
@rocks  = qw/ bedrock slate lava /;
@tiny   = ( );                      # the empty list
@giant  = 1..1e5;                   # a list with 100,000 elements
@stuff  = (@giant, undef, @giant);  # a list with 200,001 elements
$dino   = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

That last assignment gives `@quarry` the five-element list `(bedrock, slate, lava, crushed rock, granite)` , since `@tiny` contributes zero

elements to the list. In particular, it doesn't add an `undef` item to the list—but you could do that explicitly, as we did with `@stuff` earlier. It's also worth noting that an array name expands to the list it contains. An array doesn't become an element in the list, because these arrays can contain only scalars, not other arrays. The value of an array variable that has not yet been assigned is `( )`, the empty list. Just as new, empty scalars start out with `undef`, new, empty arrays start out with the empty list.

NOTE

In *Intermediate Perl*, we cover references, which lets you make what are informally called "lists of lists," among other interesting and useful structures. The perldsc documentation is worth a read.

When you copy an array to another array, it's still a list assignment. The lists are simply stored in arrays. For example:

```
@copy = @quarry; # copy a list from one array to another
```

The pop and push Operators

You *could* add new items to the end of an array by simply storing them as elements with new, larger indices.

One common use of an array is as a *stack* of information, where you add new values to and remove old values from the righthand side of the list. The righthand side is the end with the "last" items in the array, the end with the highest index values. These operations occur often enough to have their own special functions. Think of this like a stack of plates. You take a plate off the top of the stack and put plates on top of the stack (if you're like most people).

The `pop` operator takes the last element off of an array and returns it:

```
@array  = 5..9;
$fred   = pop(@array);  # $fred gets 9, @array now has (5, 6, 7, 8)
```

```
$barney = pop @array;    # $barney gets 8, @array now has (5, 6, 7)
pop @array;              # @array now has (5, 6). (The 7 is discarded.)
```

That last example uses `pop` in a *void context*, which is merely a fancy way of saying the return value isn't going anywhere. There's nothing wrong with using `pop` in this way, if that's what you want.

If the array is empty, `pop` leaves it alone (since there is no element to remove) and returns `undef`.

You may have noticed that you can use `pop` with or without parentheses. This is a general rule in Perl: as long as you don't change the meaning by removing the parentheses, they're optional. The converse operation is `push`, which adds an element (or a list of elements) to the end of an array:

```
push(@array, 0);      # @array now has (5, 6, 0)
push @array, 8;       # @array now has (5, 6, 0, 8)
push @array, 1..10;   # @array now has those 10 new elements
@others = qw/ 9 0 2 1 0 /;
push @array, @others; # @array now has those five new elements (19 total)
```

Note that the first argument to `push` or the only argument for `pop` must be an array variable—pushing and popping would not make sense on a literal list.

The shift and unshift Operators

The `push` and `pop` operators do things to the end of an array (or the right side of an array, or the portion with the highest subscripts, depending on how you like to think of it). Similarly, the `unshift` and `shift` operators perform the corresponding actions on the "start" of the array (or the "left" side of an array, or the portion with the lowest subscripts). Here are a few examples:

```
@array = qw# dino fred barney #;
$m = shift(@array);      # $m gets "dino", @array now has ("fred", "barney")
```

```
$n = shift @array;       # $n gets "fred", @array now has ("barney")
shift @array;            # @array is now empty
$o = shift @array;       # $o gets undef, @array is still empty
unshift(@array, 5);      # @array now has the one-element list (5)
unshift @array, 4;       # @array now has (4, 5)
@others = 1..3;
unshift @array, @others; # @array now has (1, 2, 3, 4, 5)
```

Analogous to `pop`, `shift` returns `undef` if you give it an empty array variable.

The splice Operator

The `push`-`pop` and `shift`-`unshift` operators work with the ends of the array, but what if you need to remove or add elements to the middle? That's where the `splice` operator comes in. It takes up to four arguments, two of which are optional. The first argument is always the array and the second argument is the position where you want to start. If you only use those two arguments, Perl removes all of the elements from your starting position to the end and returns them to you:

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 2; # remove fred and everything after
                             # @removed is qw(fred barney betty)
                             # @array is qw(pebbles dino)
```

You can use a third argument to specify a length. Read that sentence again because many people assume that the third argument is an ending position, but no, it's a length. That way, you can remove elements from the middle and leave some at the end:

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 2; # remove dino, fred
                                # @removed is qw(dino fred)
                                # @array is qw(pebbles barney betty)
```

The fourth argument is a replacement list. At the same time that you take some elements out, you can put others in. The replacement list does not need to be the same size as the slice that you are removing:

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 2, qw(wilma); # remove dino, fred
                                 # @removed is qw(dino fred)
                                 # @array is qw(pebbles wilma
                                 #                barney betty)
```

You don't have to remove any elements. If you specify a length of 0, you remove no elements but still insert the "replacement" list:

```
@array = qw( pebbles dino fred barney betty );
@removed = splice @array, 1, 0, qw(wilma); # remove nothing
                                 # @removed is qw()
                                 # @array is qw(pebbles wilma dino
                                 #               fred barney betty)
```

Notice that `wilma` shows up before `dino`. Perl inserted the replacement list starting at index 1 and moved everything else over.

`splice` might not seem like a big deal to you, but this is a hard thing to do in some languages, and many people developed complicated techniques, such as linked lists, that take a lot of programmer attention to get right. Perl handles those details for you.

Interpolating Arrays into Strings

As with scalars, you can interpolate array values into a double-quoted string. Perl expands the array and automatically adds spaces between the elements, putting the whole result in the string upon interpolation:

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # prints five rocks separated by spaces
```

There are no extra spaces added before or after an interpolated array; if you want those, you'll have to put them in yourself:

```
print "Three rocks are: @rocks.\n";
print "There's nothing in the parens (@empty) here.\n";
```

If you forget that arrays interpolate like this, you'll be surprised when you put an email address into a double-quoted string:

```
$email = "fred@bedrock.edu";  # WRONG! Tries to interpolate @bedrock
```

Although you probably intended to have an email address, Perl sees the array named `@bedrock` and tries to interpolate it. Depending on your version of Perl, you'll probably just get a warning:

```
Possible unintended interpolation of @bedrock
```

To get around this problem, you either escape the `@` in a double-quoted string or use a single-quoted string:

```
$email = "fred\@bedrock.edu"; # Correct
$email = 'fred@bedrock.edu';  # Another way to do that
```

A single element of an array interpolates into its value, just as you'd expect from a scalar variable:

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";    # "This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # same thing
```

Note that the index expression evaluates as an ordinary expression, as if it were outside a string. It is *not* variable-interpolated first. In other words, if `$y` contains the string `"2*4"`, we're still talking about element

1, not element 7, because the string `"2*4"` as a number (the value of `$y` used in a numeric expression) is just plain `2`.

If you want to follow a simple scalar variable with a left square bracket, you need to delimit the square bracket so that it isn't considered part of an array reference, as follows:

```
@fred = qw(eating rocks is wrong);
$fred = "right";               # we are trying to say "this is right[3]"
print "this is $fred[3]\n";    # prints "wrong" using $fred[3]
print "this is ${fred}[3]\n";  # prints "right" (protected by braces)
print "this is $fred"."[3]\n"; # right again (different string)
print "this is $fred\[3]\n";   # right again (backslash hides it)
```

The foreach Control Structure

It's handy to be able to process an entire array or list, so Perl provides a control structure to do just that. The `foreach` loop steps through a list of values, executing one iteration (time through the loop) for each value:

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";  # Prints names of three rocks
}
```

The control variable (`$rock` in that example) takes on a new value from the list for each iteration. The first time through the loop, it's `"bedrock"`; the third time, it's `"lava"`.

The control variable is not a copy of the list element—it actually *is* the list element. That is, if you modify the control variable inside the loop, you modify the element itself, as shown in the following code snippet. This is useful, and supported, but it would surprise you if you weren't expecting it:

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
```

```
  $rock = "\t$rock";       # put a tab in front of each element of @rocks
  $rock .= "\n";           # put a newline on the end of each
}
print "The rocks are:\n", @rocks; # Each one is indented, on its own line
```

What is the value of the control variable after the loop has finished? It's the same as it was before the loop started. Perl automatically saves and restores the value of the control variable of a `foreach` loop. While the loop is running, there's no way to access or alter that saved value. So after the loop is done, the variable has the value it had before the loop, or `undef` if it hadn't had a value:

```
$rock = 'shale';
@rocks = qw/ bedrock slate lava /;

foreach $rock (@rocks) {
  ...
}

print "rock is still $rock\n"; # 'rock is still shale'
```

That means that if you want to name your loop control variable `$rock`, you don't have to worry that maybe you've already used that name for another variable. After we introduce subroutines to you in Chapter 4, we'll show you a better way to handle that.

NOTE

That triple dot (`...`) is actually valid Perl. It was added in v5.12 as a placeholder. It compiles but is a fatal error if the program encounters it. There's a range operator that looks the same, but since this use stands alone, it's the *yada yada* operator.

Perl's Favorite Default: $_

If you omit the control variable from the beginning of the `foreach` loop, Perl uses its favorite default variable, `$_`. This is (mostly) just like any other scalar variable, except for its unusual name. For example:

```
foreach (1..10) {  # Uses $_ by default
  print "I can count to $_!\n";
}
```

Although this isn't Perl's only default by a long shot, it's Perl's most common default. You'll see many other cases in which Perl will automatically use `$_` when you don't tell it to use some other variable or value, thereby saving the programmer from the heavy labor of having to think up and type a new variable name. So as not to keep you in suspense, one of those cases is `print`, which will output `$_` if given no other argument:

```
$_ = "Yabba dabba doo\n";
print;  # prints $_ by default
```

The reverse Operator

The `reverse` operator takes a list of values (which may come from an array) and returns the list in the opposite order. So if you were disappointed that the range operator only counts upward, this is the way to fix it:

```
@fred   = 6..10;
@barney = reverse(@fred);  # gets 10, 9, 8, 7, 6
@wilma  = reverse 6..10;   # gets the same thing, without the other array
@fred   = reverse @fred;   # puts the result back into the original array
```

The last line is noteworthy because it uses `@fred` twice. Perl always calculates the value being assigned (on the right) before it begins the actual assignment.

Remember that `reverse` returns the reversed list; it doesn't affect its arguments. If the return value isn't assigned anywhere, it's useless:

```
reverse @fred;         # WRONG - doesn't change @fred
@fred = reverse @fred; # that's better
```

The sort Operator

The `sort` operator takes a list of values (which may come from an array) and sorts them in the internal character ordering. For strings, that would be in code point order. In pre-Unicode Perls, the sort order was based on ASCII, but Unicode maintains that same order as well as defining the order of many more characters. So, the code point order is a strange place where all of the capital letters come before all of the lowercase letters, where the numbers come before the letters, and the punctuation marks—well, those are here, there, and everywhere. But sorting in that order is just the *default* behavior; you'll see in Chapter 14 how to sort in whatever order you'd like. The `sort` operator takes an input list, sorts it, and outputs a new list:

```
@rocks   = qw/ bedrock slate rubble granite /;
@sorted  = sort(@rocks);        # gets bedrock, granite, rubble, slate
@back    = reverse sort @rocks; # these go from slate to bedrock
@rocks   = sort @rocks;         # puts sorted result back into @rocks
@numbers = sort 97..102;        # gets 100, 101, 102, 97, 98, 99
```

As you can see from that last example, sorting numbers as if they were strings may not give useful results. But, of course, any string that starts with `1` has to sort before any string that starts with `9`, according to the default sorting rules. And like what happened with `reverse`, the arguments themselves aren't affected. If you want to sort an array, you must store the result back into that array:

```
sort @rocks;          # WRONG, doesn't modify @rocks
@rocks = sort @rocks; # Now the rock collection is in order
```

The each Operator

Starting with v5.12, you can use the `each` operator on arrays. Before that version, you could only use `each` with hashes, but we don't show you those until Chapter 6. Every time you call `each` on an array, it returns

two values for the next element in the array—the index of the value and the value itself:

```
require v5.12;

@rocks = qw/ bedrock slate rubble granite /;
while( ( $index, $value ) = each @rocks ) {
    print "$index: $value\n";
}
```

NOTE

We used `require` here because a `use v5.12` would turn on "strict" mode. We don't tell you how to fix that until Chapter 4, so we punt here. You'll be fine after the next chapter.

If you wanted to do this without `each`, you'd have to iterate through all of the indices of the array and use the index to get the value:

```
@rocks   = qw/ bedrock slate rubble granite /;
foreach $index ( 0 .. $#rocks ) {
    print "$index: $rocks[$index]\n";
}
```

Depending on your task, one or the other may be more convenient for you.

Scalar and List Context

This is the most important section in this chapter. In fact, it's the most important section in the entire book. It wouldn't be an exaggeration to say that your entire career in using Perl will depend on understanding this section. So if you've gotten away with skimming the text up to this point, this is where you should really pay attention.

That's not to say that this section is in any way difficult to understand. It's actually a simple idea: a given expression may mean different things depending upon where it appears and how you use it. This is nothing new to you; it happens all the time in natural languages. For example, in English, suppose someone asked you what the word "flies" means. It has different meanings depending on how it's used. You can't identify the meaning until you know the *context*.

The context refers to how you use an expression. You've actually already seen some contextual operations with numbers and strings. When you do numbery sorts of things, you get numeric results. When you do stringy sorts of things, you get string results. And it's the operator that decides what you are doing, not the values. The `*` in `2*3` is numeric multiplication, while the `x` in `2x3` is string replication. The first gives you `6` while the second gives you `222`. That's context at work for you.

As Perl is parsing your expressions, it always expects either a scalar value or a list value (or void, which we don't cover in this book). What Perl expects is called the context of the expression:

```
42 + something # The something must be a scalar
sort something # The something must be a list
```

This is like spoken languages. If we make a grammatical mistake, you notice it right away because you expect certain words in certain places. Eventually, you'll read Perl this way too, but at first you have to think about it.

Even if *something* is the exact same sequence of characters, in one case it may give a single, scalar value, while in the other, it may give a list. Expressions in Perl always return the appropriate value for their context. For example, how about the "name" of an array. In a list context, it gives the list of elements. But in a scalar context, it returns the number of elements in the array:

```
@people = qw( fred barney betty );
@sorted = sort @people;  # list context: barney, betty, fred
```

```
$number = 42 + @people;  # scalar context: 42 + 3 gives 45
```

Even ordinary assignment (to a scalar or a list) causes different contexts:

```
@list = @people; # a list of three people
$n = @people;    # the number 3
```

But please don't jump to the conclusion that scalar context always gives the number of elements that would have been returned in list context. Most list-producing expressions return something *much* more interesting.

Any expression can produce a list or a scalar depending on context. So when we say "list-producing expressions," we mean those that are typically used in a list context and therefore might surprise you when they're used unexpectedly in a scalar context (like `reverse` or `@fred`).

Not only that, but you can't make any general rules to apply what you know about some expressions to others. Each expression can make up its own rules. Or, really, follow the overall rule that isn't very helpful to you: do the thing that makes the most sense for that context. Perl is very much a language that tries to do the most common, mostly right thing for you.

Using List-Producing Expressions in Scalar Context

There are many expressions that you will typically use to produce a list. If you use one in a scalar context, what do you get? See what the author of that operation says about it. Usually that person is Larry, and usually the documentation gives the whole story. In fact, a big part of learning Perl is actually learning how Larry thinks. Therefore, once you can think like Larry does, you know what Perl should do. But while you're learning, you'll probably need to look into the documentation.

Some expressions don't have a scalar-context value at all. For example, what should `sort` return in a scalar context? You wouldn't need to sort a list to count its elements, so until someone implements something else, `sort` in a scalar context always returns `undef`.

Another example is `reverse`. In a list context, it gives a reversed list. In a scalar context, it returns a reversed string (or reversing the result of concatenating all the strings of a list, if given one):

```
@backwards = reverse qw/ yabba dabba doo /;
    # gives doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
    # gives oodabbadabbay
```

At first, it's not always obvious whether an expression is being used in a scalar or a list context. But trust us, it *will* become second nature for you eventually.

Here are some common contexts to start you off:

```
$fred = something;            # scalar context
@pebbles = something;         # list context
($wilma, $betty) = something; # list context
($dino) = something;          # still list context!
```

Don't be fooled by the one-element list; that last one is a list context, not a scalar one. The parentheses are significant here, making the fourth of those different than the first. If you assign to a list (no matter the number of elements), it's a list context. If you assign to an array, it's a list context.

Here are some other expressions you've seen, and the contexts they provide. First, some that provide scalar context to *something*:

```
$fred = something;
$fred[3] = something;
123 + something
something + 654
if (something) { ... }
while (something) { ... }
$fred[something] = something;
```

And here are some that provide a list context:

```
@fred = something;
($fred, $barney) = something;
($fred) = something;
push @fred, something;
foreach $fred (something) { ... }
sort something
reverse something
print something
```

Using Scalar-Producing Expressions in List Context

Going this direction is straightforward: if an expression doesn't normally have a list value, the scalar value is automatically promoted to make a one-element list:

```
@fred = 6 * 7; # gets the one-element list (42)
@barney = "hello" . ' ' . "world";
```

Well, there's one possible catch. Since `undef` is a scalar value, assigning `undef` to an array doesn't clear the array. The better way to do that is to assign an empty list:

```
@wilma = undef; # OOPS! Gets the one-element list (undef)
    # which is not the same as this:
@betty = ( );    # A correct way to empty an array
```

Forcing Scalar Context

On occasion, you may need to force scalar context where Perl is expecting a list. In that case, you can use the fake function `scalar`. It's not a true function because it just tells Perl to provide a scalar context:

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
```

```
print "I have ", @rocks, " rocks!\n";        # WRONG, prints names of rocks
print "I have ", scalar @rocks, " rocks!\n"; # Correct, gives a number
```

Oddly enough, there's no corresponding function to force list context. It turns out you almost never need it. Trust us on this too.

<STDIN> in List Context

One previously seen operator that returns a different value in an array context is the line-input operator, `<STDIN>`. As we described earlier, `<STDIN>` returns the next line of input in a scalar context. Now, in list context, this operator returns *all* of the remaining lines up to the end-of-file. It returns each line as a separate element of the list. For example:

```
@lines = <STDIN>; # read standard input in list context
```

When the input is coming from a file, this will read the rest of the file. But how can there be an end-of-file when the input comes from the keyboard? On Unix and similar systems, including Linux and macOS, you'll normally type a Ctrl-D to indicate to the system that there's no more input; the special character itself is never seen by Perl, even though it may be echoed to the screen. On DOS/Windows systems, use Ctrl+Z instead. You'll need to check the documentation for your system or ask your local expert if it's different from these.

NOTE

There's a bug affecting some ports of Perl for DOS/Windows where the first line of output to the terminal following the use of Ctrl+Z is obscured. On these systems, you can work around this problem by simply printing a blank line (`"\n"`) after reading the input.

If the person running the program types three lines, then presses the proper keys needed to indicate end-of-file, the array ends up with three

elements. Each element will be a string that ends in a newline, corresponding to the three newline-terminated lines entered.

Wouldn't it be nice if, having read those lines, you could `chomp` the newlines all at once? It turns out that if you give `chomp` an array holding a list of lines, it will remove the newlines from each item in the list. For example:

```
@lines = <STDIN>; # Read all the lines
chomp(@lines);    # discard all the newline characters
```

But the more common way to write that is with code similar to what you used earlier:

```
chomp(@lines = <STDIN>); # Read the lines, not the newlines
```

Although you're welcome to write your code either way in the privacy of your own cubicle, most Perl programmers will expect the second, more compact, notation.

It may be obvious to you (but it's not obvious to everyone) that once these lines of input have been read, they can't be reread. Once you've reached end-of-file, there's no more input out there to read.

And what happens if the input is coming from a 4 TB logfile? The line-input operator reads all of the lines, gobbling up lots of memory. Perl tries not to limit you in what you can do, but the other users of your system (not to mention your system administrator) are likely to object. If the input data is large, you should generally find a way to deal with it without reading it all into memory at once.

Exercises

See "Answers to Chapter 3 Exercises" for answers to these exercises:

1. [6] Write a program that reads a list of strings on separate lines until end-of-input and prints out the list in reverse order. If the input comes from the keyboard, you'll probably need to signal the end of the input by pressing Ctrl-D in Unix or Ctrl+Z on Windows.
2. [12] Write a program that reads a list of numbers (on separate lines) until end-of-input and then prints for each number the corresponding person's name from the following list. (Hardcode this list of names into your program. That is, it should appear in your program's source code.) For example, if the input numbers were `1`, `2`, `4`, and `2`, the output names would be `fred`, `betty`, `dino`, and `betty`:

```
fred betty barney dino wilma pebbles bamm-bamm
```

3. [8] Write a program that reads a list of strings (on separate lines) until end-of-input. Then it should print the strings in code point order. That is, if you enter the strings `fred`, `barney`, `wilma`, `betty`, the output should show `barney betty fred wilma`. Are all of the strings on one line in the output or on separate lines? Could you make the output appear in either style?