13 Lists

Chapter 12 was rather glib about *lists* – sequences of items separated by commas and surrounded by round brackets. That notation appeared in the last chapter both with print statements:

```
print($countyNames[12], "\t", $countyPops[12], "\n");
```

and also for initializing the various values of an array:

```
@numbers = (1, 2, 3, 4, 5);
```

In previous chapters, `print` was always followed by a single string or variable name. What exactly does it mean to link a list of items with commas and surround it with round brackets?

Lists in Perl are treated by some textbook writers as another type of data structure alongside scalars, arrays, and hashes (which we have not covered yet). For instance, a `print` statement in general takes a list of items to print – where just one item is printed, this is a special case (though a very frequent one). However, unlike scalars, arrays, and hashes, a list is not something that has its own variable name (so there is no prefix symbol for lists, parallel to `$`, `@`, and `%`). We can form a list by naming its elements, separated by commas, between a pair of round brackets, but we cannot refer to it using a single symbol. Really, lists are a variety of array, which contains elements in a fixed sequence, but which is unnamed and unnamable, and which does not continue in existence after the statement which creates it (as an array does).[21]

If we want to turn a list into a named entity with continuing existence, we can assign it as the value of an array; as we have seen, this is the easiest way to populate an array with elements. We can also use a list of variables on the left of an equals-sign in order to assign values to several variables at once:

```
($a, $b, $c) = (15, 20, 25);
```

is a good alternative to:

```
$a = 15;
$b = 20;
$c = 25;
```

Normally when assigning a list of values to a list of variables there would be the same number of items in the two lists. A statement such as:

```
($a, $b, $c) = (15, 20);
```

would destroy any value that `$c` had previously, without giving it a new value. And the statement:

```
($a, $b) = (15, 20, 25);
```

simply makes no use of the value 25.

Strictly, if the list on the left-hand side of an assignment includes an array variable, because an array can contain any number of items there can be more values in the right-hand list than variables in the left-hand list. But we have to be very careful here. One might imagine that

```
($mother, @twins, $child3, $child4)
  = ("Mummy", "Topsy", "Tim", "Tansy", "Teddy");
```

would lead to the following assignments:

```
$mother       "Mummy"
@twins        "Topsy", "Tim"
$child3       "Tansy"
$child4       "Teddy"
```

But it won't; it will actually give these assignments:

```
$mother       "Mummy"
@twins        "Topsy", "Tim", "Tansy", "Teddy"
```

and neither `$child3` nor `$child4` will be assigned any values. The first array variable in the left-hand list will always mop up any values on the right that have not already been used. It is best to avoid this problem, by including only scalar variables in lists on the left-hand side of equals signs. The statements:

```
($mother, $child3, $child4) = ("Mummy", "Tansy", "Teddy");
@twins = ("Topsy", "Tim");
```

achieves the desired set of scalar and array variable assignments in a way that leaves no room for confusion.

If there are array variables in a list on the *right*-hand side of an assignment, another pitfall arises. Lists "flatten out" any multi-element variables within them into their individual elements. Suppose we have given the arrays `@words` and `@numbers` the same values that we gave them in section 12.3:

```
@words = ("the","quality","of","mercy","is","not","strained");
@numbers = (1, 2, 3, 4, 5);
```

Then the statement

```
@wordsAndNums = (@words, @numbers);
```

will not create a two-member array; `@wordsAndNums` is now a twelve-member array, with `$wordsAndNums[0]` being `"the"` and `$wordsAndNums[11]` being 5, because the bracket-and-comma notation to the right of the equals sign has flattened the separate arrays into one long list of scalars. (We shall see in chapter 15 that we can have arrays whose members are themselves arrays, but this is not the way to create them.)

All in all, although in one way it is reasonable to explain lists as temporary, nameless arrays, this really understates the difference between the two concepts. Rather than pursue the issue in more detail, it will be best to leave the topic here and to suggest that readers avoid being too imaginative in how they deploy lists within their programs. Using lists in the ways they are used in our examples should not create problems, but doing other kinds of thing with lists can lead to unexpected results. We can always assign a list of values to a named array and work with that; the behaviour of arrays is easier to predict.