15 Two-dimensional tables

In our work with the county data, I said that setting up two arrays, one for the county names and one for their populations, was not the ideal approach. The trouble is that there is nothing but numerical position in the respective sequences to tie individual population figures to the correct names. This is risky. Code-chunk (13), which set the arrays up, will never put a population figure in a wrong cell, differently-numbered from the cell containing the corresponding name; but, in real-life programming, data sets often have to be changed and updated after they are initially created, and we have set up a system here where it would be all too easy for the `@countyNames` and `@countyPops` arrays to get out of synch with one another. It *shouldn't* happen – but, with software, if something can go wrong then sooner or later it probably will.

We are really dealing with a single set of data: on paper it would be one two-dimensional table, with many rows (one for each county) and two columns. So it ought to be coded as a single array, `@counties`, within which an individual county's name and population will be a single data element. And that is easy to do.

Until now, array elements have always been scalars – numbers, or strings. But in Perl they can be anything we like, including arrays.[24] What we want to do is to set up a `@counties` array, within which each row is a two-element array containing a name and a population figure. Then we have eliminated the risk of parallel arrays getting out of synch with one another. We will have mapped the two-dimensional shape of the table on paper into an electronic equivalent.

Within an array of arrays, the value in the *j*'th column of the *i*'th row will be identified by two pairs of square brackets:

```
$arrofarr[i][j]
```

So, to create the `@counties` array, we need to replace lines 16 and 17 of (13) with the following:

```
$counties[$i][0] = $name;
$counties[$i][1] = $population;
```

Thus our new version of code-chunk (13) will be:

(18)

```
1   open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2   $i = 0;
3   while ($a = <INFILE>)
4     {
5     if ($a !~ /\S/) {;}
6     elsif ($a !~ /^\s*(\S.*\S)\s+(\d+)\s*$/)
7       {
8       die "bad input line:  $a\n";
9       }
10    else
11      {
12      $name = $1;
13      $population = $2;
14      $name = join("_", split(/\s+/, $name));
15      $population *= 1000;
16      $counties[$i][0] = $name;
17      $counties[$i][1] = $population;
18      ++$i;
19      }
20    }
21  close(INFILE);
```

Now that we have our two-dimensional table encoded as an array of arrays, we can pick out an individual scalar data item from it using two pairs of square brackets:

```
print "$counties[40][0]\n";
```

```
Isle_of_Wight
```

```
print "$counties[40][1]\n";
```

```
97000
```

However, I said in the footnote above that Perl only behaved *as if* it included arrays of arrays. What we have actually done with (18) is not exactly what we seem to have done, and you need to be aware of this so that you don't get puzzled when other things you might think of doing with arrays of arrays fail to work.

The elements in the rows of `@counties` are not actually arrays, i.e. sets of data values; they are single items called *references*. We can think of a Perl reference as a sort of "pointer", telling Perl where to find some other item (in this case, where to find another array).[25] In reality, Perl arrays can be only one-dimensional. But, because array elements can be references to sets of data values, a Perl array can *appear* to be multi-dimensional.

One way in which this difference becomes noticeable in practice is that, if you already have an array, you cannot assign it as a whole to be an element of a larger array. So, for instance, you might have thought that rather than putting county name and county population separately into separate cells of the two-dimensional `@counties` array, we could instead for each county have created a two-element array, and then assigned that array as a row of `@counties`. In other words, we might have written the following instead of 18.16–17:

```
16  @county = ($name, $population);
17  @counties[$i] = @county;     # NO!
```

That won't work. Line 16 will create a two-element array `@county` containing a string and a number; but line 17 will not make that array the value of an element of `@counties`. If you are running Perl with warnings (using `perl -w`) you will be warned that line 17 ought to begin with `$`, not `@`, because any element of an array must be a scalar; and if you try printing out, say, `$counties[4][0]`, no county name will emerge.

Provided you stick to building up an array of arrays individual cell by individual cell, though (as we did in (18), where line 16 put a string into column 0 and line 17 put a number into column 1 for each county), Perl is organized so that you do not need to worry about the machinery which creates the appearance of an array whose rows are themselves multi-element arrays. And in this book we shall not worry about it. References are probably the most important Perl topic which this book deliberately avoids discussing in detail – the reason being that beginners usually find it a very confusing topic, and we can achieve quite a lot in Perl without learning about references.

If you progress with Perl, sooner or later you will need to learn about references and "dereferencing". These topics are needed for other purposes as well as multi-dimensional arrays. But I said at the beginning that serious Perl users will need to go on to consult more comprehensive manuals than this brief beginners' textbook. Some experts would argue that even beginners need to learn about references. I believe it will be better to leave them aside, until the reader has grown in confidence working with the Perl constructions that this book does cover. From now on, then, we shall pretend that Perl does have true arrays of arrays, but we shall be careful always to put values into arrays of arrays, and get values out, one individual cell at a time.

That is a constraint on the arrays-of-arrays concept in Perl. In another respect, though, Perl arrays of arrays are *more* flexible than in some other programming languages. Our array `@counties`, although it is composed of arrays rather than of scalars (really, "appears to be composed …" – but I shall stop saying that, and write from now on as if appearance were reality), nevertheless is what one might call "regular". The value in *each* row is always an array containing exactly two elements, one string and one number. In some languages, multi-dimensional arrays *must* be regular. But nothing in Perl requires this. It is perfectly possible to create an array in which rows contain different numbers of elements. (It is even possible to create an array in which some rows contain individual scalars, other rows contain arrays with different numbers of elements, and some of those elements could in turn themselves be arrays.)

How often a thoroughly chaotic array like that last one would actually be useful in practice is another question. Very rarely, I imagine. But mildly irregular arrays often are useful. Although we have introduced the concept "array of arrays" through an example which is totally regular, bear in mind that this is just a special case.