12 Arrays

12.1 Tables with numbered cells

So far, all our variables have been *scalars*, with names having `$` as their prefix. We said that apart from scalars, for individual data items, Perl has two other types of variable, for organized sets of items: *arrays*, and *hashes*, with prefixes `@` and `%` respectively. The array type is found in most modern programming languages. The hash type is less widespread, more of a Perl speciality (though there are other languages which include it); we shall look at hashes in chapter 17.[17]

An array is like a table, which as a whole has a single name, and within which different pieces of data occupy successive numbered rows. (Tables can have many columns as well as many rows, but for the moment think of an array as a table with a single column of cells.) This being the computing world, the initial cell of the table is numbered zero, rather than one.

Consider again our county-population data. So far, we have seen how to read these data in from a rather messy file, and print them out again as a reformatted, neater file. But we might want to hold the data within our program, so that later activities within the program can refer to various of these data items. One way to do this will be to use the input lines to build up a pair of arrays, `@countyNames` and `@countyPops`, so that `@countyNames` holds the county names in a fixed sequence, and `@countyPops` holds the population figures in the corresponding sequence. Then we can ask a question like "What is the name of county number 11?" by writing:

```
print "$countyNames[11]\n";
```

to which the answer will be:

```
Essex
```

The answer will be `Essex` rather than `Gloucestershire`, because Bedfordshire will be county number 0 not number 1 – see above. More important, notice the dollar sign in the print statement. We have introduced the idea of arrays and said that `@countyNames` is an array, hence its name begins with the `@` symbol. But when we put square brackets after the array name in order to pick out an individual member of the array, the prefix changes to the dollar sign.

This is an odd feature of Perl which beginners usually find confusing, so it deserves a little discussion. Logically, one might expect that if, say, `@fruits` is an array, then the *n*th member of `@fruits` would be identified as `@fruits[n]`, so that one could write a statement like:

```
print @fruits[5];      # wrong!
```

That isn't how Perl works. Because item 5 of the `@fruits` array is an individual item, one has to write:

```
print $fruits[5];
damson
```

– even though the symbol `$fruits` on its own, without following square brackets, refers to nothing.

Apparently, when Perl was created, the inventor believed that users would find this the more natural thing. He was wrong there. It is generally recognized now that this decision was unfortunate (and we are promised that when Perl 6 eventually replaces the current version Perl 5, the decision will be reversed). But we shall be living with Perl 5 for a long time yet, so we just have to get used to this oddity. That is not particularly difficult to do, once we face up to the fact that the rule is different from what most of us instinctively expect.

12.2. An example

Let's now adapt the code (12) which we used to print out the tidied-up county data, so that it instead creates a pair of arrays containing the data: an array `@countyNames` containing the county names, and an array `@countyPops` containing the population figures in the corresponding rows. (This is not the ideal approach – we shall see a better one shortly; but it is the easiest way to start with.)

Where stretches of code do not change, instead of copying them out here I shall just write `[as before]`, so that we can focus on the areas of code which do change.

(13)

```
1  open(INFILE, "<../data/pops/countyData.txt")
      or die "Cannot open input file\n";
2  $i = 0;
3  while ($a = <INFILE>)
4    {
5    if [as before]
6    elsif [as before]
     ⋮
10   else
11     {
12     $name = $1;
13     $population = $2;
14     $name =~ s/(\S)\s+(\S)/$1_$2/g;
15     $population *= 1000;
16     $countyNames[$i] = $name;
17     $countyPops[$i] = $population;
18     ++$i;
19     }
20   }
21 close(INFILE);
```

In line 13.1 we open the input file the same as before, but this time we shall not be printing out a reformatted file, so there is no `open(OUTFILE, ...)`. A new thing we do need to do before embarking on the `while` loop is to set up an index variable, `$i`, which will keep count of positions within the array; on each pass through the loop it will ensure that the name and population of the current county are placed in corresponding rows in the two arrays, and at the end of the loop `$i` will be increased by 1 ready for the next pass.

The initial row of any array is row zero, so 13. 2 sets `$i` to that value. Of course, if human users of the software are going to find it really awkward to think about "county number 0", nothing prevents us giving `$i` the initial value 1, so that Bedfordshire will become county no. 1, and row 0 in both arrays will be "wasted" (it is not like wasting real physical resources!) But if you choose to do that, you will have to remember that you have done it, and make sure that in the future you never carelessly try to use the value of `$countyNames[0]` or `$countyPops[0]` – which would be zero or rubbish values. It is probably easier in the long run to get used to the idea that the initial item of any array is item zero.[18]

Within the `while` loop, there is no change to the way that `if` and `elsif` deal with blank lines and lines with ill-formed contents, or to how the temporary variables `$1` and `$2` are replaced with `$name` and `$population` in the `else` block. Line 13.14 replaces spaces in county names with underline characters as before. But we do not want the population figures to be given commas grouping them into threes, as in the old line 12.16. That was done to turn a number into a string easily grasped by human eyes; but in our `@countyPops` array we shall want the populations to be actual numbers that the machine can calculate with, so that subsequent program code will be able to use the numerical data. So our new line 13.15 simply multiplies the figures taken from the input file by 1000, so that counts of thousands become counts of individuals, rounded to the nearest thousand.

Then, 13.16 puts the current county name in row `$i` of `@countyNames`, 13.17 puts the current county population in the corresponding row of `@countyPops`, and 13.18 increments `$i`, ready for the next pass through the loop.

Notice that we did not have to "warn" Perl that we were going to set up arrays `@countyNames` and `@countyPops` – as soon as a value is first assigned to some row of an array, if the array does not yet exist it is automatically created without further ado. (In this case we did not even mention the names `@countyNames` and `@countyPops` with `@` prefixes; but the square brackets in e.g. `$countyNames[$i]` tell Perl that what precedes them is an array name.)

Line 13.21 closes INFILE as before, but since we never opened an output file, of course we do not now have to close it.

The chunk of code (13) is not intended as a complete program on its own. That does not mean that it lacks some particular elements which Perl requires to occur in any complete program. It doesn't; a "program" in Perl is simply a sequence of statements, so we could call (13) a program if we wanted to. But as a complete program (13) would be no practical use, because it produces no output. It is intended as the beginning of a program, which uses the countyData file in order to create data structures (the arrays `@countyNames` and `@countyPops`) which can be used by code that will be added later in the program, in order to do whatever we may want to do with the data.

Obviously, we can print out individual values:

```
print($countyNames[36], "\t", $countyPops[36], "\n");
```

```
East_Sussex	710000
```

Often, we shall want to write code to do something to *each* element of an array in turn, and chapter 14 will show us how to do that.

(Since (13) is not intended as a complete program, but it seems too long to call a "snippet", I shall call it a "code-chunk". Later chapters will extend (13) into programs which produce output.)

12.3 Assigning a list to an array

Code-chunk (13) put elements into the `@countyNames` and `@countyPops` arrays one by one; but sometimes we will want to set up an array and initialize its contents in one go. The easiest way to do that is to put on the right of the equals sign a list of the array members, separated by commas and with round brackets around the list:

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

Alternatively we could of course put the items in one by one, e.g.:

```
$words[0] = "the";
$words[1] = "quality";
    ⋮
$words[6] = "strained";
```

but the bracket-and-comma notation takes less typing.

We can display the contents of an array by giving an array name as argument to a `print` statement. But, if we do this in the obvious way, what gets displayed are the array elements and nothing else. Thus, if `@words` and `@numbers` have the values as above:

```
print(@words, "\n", @numbers, "\n");

thequalityofmercyisnotstrained
12345
```

This is perfectly logical, but not very user-friendly. The trick is to include the array names within the double quotation marks, in which case the array elements are displayed separately:

```
print "@words\n@numbers\n";

the quality of mercy is not strained
1 2 3 4 5
```

12.4 Adding elements to and removing them from arrays

We have seen how to create an array and populate it with a long list of values. But often, we want to work with individual items of an array. We already know how to do that using the square-bracket notation. Continuing with the array `@words` as above, the statement:

```
$a = $words[3];
```

will assign the string `"mercy"` to `$a`, leaving `@words` unchanged. The statement:

```
$words[3] = "generosity";
```

will change element 3 of `@words` from `"mercy"` to `"generosity"`, without altering the number of elements in `@words`.[19]

Quite commonly, though, we want to add an element or elements to an array, increasing its length; or to remove element(s) from an array, using their values and leaving the array shorter.

The functions `push()` and `pop()` respectively add elements to and remove elements from the back of an array. The functions `shift()` and `unshift()` respectively remove elements from and add elements to the front of an array. The "remove" functions `pop()` and `shift()` remove one item at a time from the respective array-end and return it as their value; but the "add" functions `push()` and `unshift()` can be used to add either a single item or a list of items.

This is most easily illustrated by example. In code-snippet (14), the right-hand column shows what the array `@colours` contains after each successive statement is executed:

(14)

```
1  @colours = ("red", "green");         red green
2  $colour1 = shift(@colours);          green
3  push(@colours, "blue", "grey");      green blue grey
4  $colour2 = pop(@colours);            green blue
5  unshift(@colours, "red", "black"); red black green blue
6  print "$colour1\t$colour2\n";

   red grey
```

(Notice that when a list of items is "unshifted" onto the front of an array, the items end up in the array in the same order as they were in the list. Line 14.5 does not first move `"red"` onto the front of `@colours` and then move `"black"` into the new front position before `"red"`.)

12.5 Other operations on arrays

We can create a shorter array as a slice from the middle of a longer array:

```
@words = ("the","quality","of","mercy","is","not","strained");
@phrase = @words[1..3];
print "@phrase\n";

quality of mercy
```

The word *the* is element 0 of `@words`, so *quality* is element 1; note the use of two dots (rather than three dots as in ordinary English) to separate the end-points of the slice.

The functions `split()` and `join()` convert between strings and arrays. The former makes a string into an array of strings; it standardly takes two arguments: a pattern, between slashes (as in pattern-matching), and the string to be split – the pattern is used to identify places where the string should be split. Thus:

```
$Rev = "At Flores in the Azores Sir Richard Grenville lay";
@words = split(/ /, $Rev);
print($words[1], "\t", $words[4], "\n");

Flores  Azores
```

Here, the pattern `/ /` (i.e. a single space character between slashes) means "treat spaces as places to split".[20] If the pattern were `//`, the empty pattern, then the string would be split into its individual characters, including spaces.

Conversely, `join()` links the elements of an array into a single string, with the array elements separated by the first argument:

```
@words = ("the","quality","of","mercy");
$colonString = join(":", @words);
print($colonString, "\t", length($colonString), "\n");

the:quality:of:mercy    20
```

The functions `split()` and `join()` give us an alternative to line 14 of (13) as a way to replace whitespace in a string by single underline characters. We could change 13.14 to:

```
@nameparts = split(/\s+/, $name);
$name = join("_", @nameparts);
```

– i.e. treat sequences of one or more whitespace characters as places to split the line, then join the resulting array `@nameparts` using the underline character. This two-line sequence can be collapsed into one line:

```
$name = join("_", split(/\s+/, $name));
```

– here the second argument to `join()` is whatever results from applying `split()` to its arguments. Even as a single line, this is no shorter than the original line 13.14 – it is rather longer, in characters; but its logic is easier to grasp.

(I make no apology for the original version of 13.14, though: that offered a useful illustration of the workings of the Perl pattern matcher. Often one *must* use the pattern matcher – no convenient alternative using built-in functions will do the job.)

A particularly powerful built-in function applicable to arrays is `map()`, which produces a new array by applying a specified operation to each element of an existing array. The first argument to `map()` defines the operation to apply, as an expression within which the element to which it is applied is represented by the symbol `$_`. The second argument gives the array whose elements are to be operated on. Thus:

(15)

```
@numbers = (1, 2, 3, 4, 5);
@words = ("end","of","chapter","twelve");
@squares = map($_**2, @numbers);
@capWords = map(uc($_), @words);
print("@squares\t@capWords\n");

1 4 9 16 25   END OF CHAPTER TWELVE
```