17 Hash tables

17.1 Tables indexed by strings

An array is a table of data where the rows are identified by their numerical position in sequence. A *hash table* (or "hash" for short), with `%` rather than `@` prefixed to its name, is like an array but the rows are indexed not by numbers but by character-strings. With a hash, it does not make sense to talk about "the sixth row" or "the initial row"; but, if we chose to use, say, names of fish as the *keys* indexing the rows of a hash, then we could ask what was in the "pike" row or the "mackerel" row.

(The values *within* the rows of a hash table, like those in the rows of an array, can be any items you like, numbers or strings; it is the way that rows are identified which creates the contrast between arrays and hashes.)

In an introductory textbook the main difficulty about hashes is not explaining the mechanics of how they work, but making clear what their point is – why anyone would want to use them. For practical reasons, the programming examples included in textbooks are normally little "toy" examples; but hashes are most useful when they contain not just a dozen or two dozen data items but hundreds, or thousands, of rows of data.

So let's consider an example like that. We shan't be able to inspect the entire contents of the hash, but we will see how it is created and, once it exists, how it is used; and because it is a large enough example to be realistic we will be able (I hope) to see why hashes are a useful thing to have available. Along the way we shall encounter one or two other, much simpler but important features of Perl that we happen not to have met yet.

As background to our hypothetical hash table, let us suppose that we are interested in the vocabulary of advertising copy. Perhaps we want to monitor trends in the choice of words used in marketing, so that we can make our own advertising come over as fresh and up-to-date, or so that we can draw links between the language of advertising and other social trends. Never mind why exactly we want to study this topic; the point is that we will suppose we have files containing a mass of examples of advertising copy, and we want to process these into a table which, for any word found in the files, gives a count of the number of occurrences of that word.[30]

The table will be a hash table; we'll call it `%adWords`. The words themselves will be the "keys" indexing the rows of the table; the contents of a row will be the frequency within our data of the word which is the key of the row. So for instance the *bargain* row might contain the value 546, meaning that the word *bargain* occurred 546 times in the files from which the table was compiled.

Suppose our program has created a table like `%adWords`. Then the way to extract an individual data item such as the one just mentioned will be to use curly brackets round the key, corresponding to square brackets round a row-number in the case of an array:

```
print $adWords{"bargain"};
```

546

The data item is a scalar, so a dollar sign appears at the beginning of `$adWords{"bargain"}` even though `%adWords` as a whole takes the `%` prefix. By now, this is what we expect in Perl.

Notice that there is no way that we can ask for, say, the contents of row 13 of `%adWords`. If we wrote `$adWords{"13"}` that would give us the frequency of the word *13*; and `$adWords{13}`, or `$adWords[13]`, would just be errors. We shall see, later, how it is possible to work through each row of a hash table in turn; it is not done by starting with row 0 and moving on to row 1, row 2, etc. – with a hash these phrases are meaningless. Hashes are useful in situations where, much of the time, one is looking at single data items within a large table, rather than working through the entire table.

17.2 Creating a hash

So how would we create the `%adWords` hash from our text files?

To save time I'll assume that we have arranged a filehandle so that the expression `<INFILE>` will read a line of text in from our data files, until there is nothing left to read in. After reading in any line, we will want to split it into an array of words separated by spaces (an array, not a hash); and then we'll work through the array using the individual words to increment the values in the corresponding rows of `%adWords`.

Of course, if we split a line of everyday text on whitespace, the units we get will not all be "words" exactly – some of them might have punctuation at beginning or end (or both), and indeed some elements surrounded by whitespace might consist wholly of punctuation (dashes are often written with spaces on either side). We don't want, say, `bargain!` or `"bargain` (or `Bargain`) to be counted separately from `bargain`, so let's say that before adding any word into the hash we'll remove any punctuation found at either end (and ignore a "word" which contains only punctuation marks), and we'll reduce all alphabetic characters to lower case. Also, as with `avewordLen()` in chapter 16, we'll ignore words containing digits – any number can appear in a price, such as *99p* or *£279*, so it would not be very meaningful to keep count of the prices that happen to occur in our data.

The following code will do to read in the lines of data and get the words ready to add into the hash table; line 26.11 does the work of actually building the hash table with the help of a user-defined function, which we shall look at below.

(26)

```
1   while ($line = <INFILE>)
2     {
3     @words = split(/\s+/, $line);
4     foreach $word (@words)
5       {
6       if ($word =~ m/\w/ and $word !~ m/\d/)
7         {
8         $word =~ s/^\W*(\w)/$1/;
9         $word =~ s/(\w)\W*$/$1/;
10        $word = lc($word);
11        updateAdWords($word);
12        }
13      }
14    }
```

Line 26.3 splits an input line into words separated by one or more whitespace characters. Then 26.4–14 work through each element of the resulting array in turn, using `$word` to refer to the current array item. Line 26.6 checks that `$word` contains at least one "word character" and no digit – if it fails these tests, for instance because it is all punctuation, nothing further will happen in this pass of the `foreach` loop (and Perl will move on to consider the next word). Line 26.8 deletes any sequence of nonword characters at the beginning of `$word`, and line 26.9 does the same at the end of `$word`. Line 26.10 reduces any capitals in the remaining string to lower case. Then 26.12 calls the subroutine `updateAdWords()`, which uses `$word` to update the `%adWords` hash table.

When a word is used to update `%adWords`, there are two possibilities: either this is a word which `%adWords` has not seen before, in which case a row needs to be created for it in the hash table with the value 1; or there is already a row for that word, in which case the value in that row needs to be increased by one. In the definition of the function it is simpler to take those alternatives in the opposite order, using the built-in function `exists()` which checks whether a given hash contains a row with a given key:

(27)

```
1   sub updateAdWords
2     {
3     my $entry = shift(@_);
4     if (exists($adWords{$entry}))
5       {
6       ++ $adWords{$entry};
7       }
8     else
9       {
10      $adWords{$entry} = 1;
11      }
12    }
```

(Strictly speaking, there is a third possibility: the first time `updateAdWords()` is called, the hash `%adWords` will not yet exist. But as soon as Perl hits 27.10 for the first time, it will create `%adWords` without further ado, as well as creating a row for the current word within it.)

And that is all there is to it, so far as creating the hash table is concerned. After (26) has run through all the data so that `<INFILE>` has no further lines of text to deliver, `%adWords` will have a row for each distinct word in the data; the key of the row will be the word and the value will be the number of times that word occurs.

If we want to delete an individual entry from a hash table, the built-in function `delete()` does this:

```
delete $adWords{"bargain"};
```

You might perhaps wonder why hashes have a `delete()` function but there is no equivalent for arrays. What would an equivalent do in the case of an array, though? When a table is indexed by row numbers, one cannot just make row 15 (say) go away – we can remove the contents of the row, but the row will still be there between rows 14 and 16. With a hash, on the other hand, after the command `delete $adWords{"bargain"}` is executed, it is as if `%adWords` had never had a `bargain` row in the first place.

17.3 Working through a hash table

We saw above that the typical use of a hash is to extract the value for an individual key; and we do that by placing the key within curly brackets. But, sometimes, we do want to work through every entry in a hash table in turn. As an exercise, we might add up all the values in `%adWords` entries in order to get a count of the total number of words (word-tokens) in our data files.

We cannot apply `foreach` directly to a hash; `foreach` applies only to arrays. For hashes, there is a related term `each`. But `each` does not mean quite what you might expect; its usage is rather counterintuitive. If `%hash` is a hash table, the expression `each(%hash)` really means "the next key/value pair in `%hash`, if there is another one". So we can go about totalling the values in `%adWords` this way:

(28)

```
1  $total = 0;
2  while (($key, $value) = each(%adWords))
3    {
4    $total += $value;
5    }
6  print "$total\n";
```

Line 28.2 takes successive key/value pairs from `%adWords` and makes the members of the pair the values of the variables `$key` and `$value`; `$key` is not needed, but `$value` is used to increment `$total`.

In 28.2, assigning an array (in this case the two-element array returned by `each`) to a list of variables within round brackets is a neat way of making multiple assignments with one equals-sign. We could alternatively have invented a variable name to contain the successive key/value pairs. That is, we could have written 28.2 as, say, `while (@kvpair = each(%adWords))`, in which case line 28.4 would have needed to become `$total += $kvpair[1]` (i.e. the second element of `@kvpair` – the key is `$kvpair[0]`). But the way we have done things in (28) seems clearer; there is no point in turning the key/value pair into an array with a name of its own, rather than a nameless list, unless that array will later be used in ways that it is not used here.

There are other ways to work through the full set of entries in a hash table. Thus, the built-in functions `keys()` and `values()` return arrays containing respectively all the keys, and all the values, in a hash named as argument. Since the result of `values()` is an array, we can use `foreach` with it:

(29)

```
1  $total = 0;
2  foreach $value (values(%adWords))
3    {
4    $total += $value;
5    }
6  print $total;
```

17.4 Advantages of hash tables

We have seen that `each` returns successive key/value pairs from a hash, and `keys()` and `values()` return the respective items as an array, which is a sequentially ordered structure. So it may seem that I have misled readers by saying that there is no concept of "position of a row in numerical sequence" for hash tables. In what order are items delivered by `each`, `keys()`, and `values()`?

The answer is, to all intents and purposes in a random order. Inside the computer the rows of a hash table must be in some order or other, but there will not be any obvious logic to the order. In particular, the order of the rows *will not reflect the sequence in which entries were added to the hash*.

Normally, this book does not "peer under the bonnet" to discuss how programming structures are actually implemented within the machine. At this level, to do that would commonly be confusing rather than helpful. In the case of hash tables, though, it may be worth making an exception and explaining just a little about how they are implemented and why they store elements in an apparently random sequence, in order to help readers appreciate the "point" of including hashes in a programming language.

If a hash table is large, the process of looking up individual items in it is much more efficient than in the case of an array. To look up an item in an array, the machine must begin at row zero and inspect successive rows until the answer is found – on average half of the entire array has to be checked. In a hash, an item is placed in a row having some arithmetic relationship to its key. For instance, if the hash has 1000 rows available, then to decide which row to use for key *X* the rule could be to multiply together the ASCII codes for the characters of *X* (a = 97, b = 98, etc.) and to use the last three digits of the large resulting number to place the item – or, if that particular row happens already to be taken for a different key, use the next empty row. Then, when an item is looked up, by making the calculation on the key string the machine can usually go straight, or almost straight, to the correct row even in a huge table. For many real-life applications, this is an important benefit.[31]

Efficient lookup is probably the most important reason for including hash tables in a programming language. But also, it can be easier for a programmer to think in terms of tables indexed by meaningful strings than tables indexed by numbers. Consider again our table of county data. At present, the only datum included for a county, apart from its name, is its population. But we might want to enlarge the table to include further numerical information – say, the area of the county in acres, and its property-tax base in pounds sterling (what back in 1966, the year to which our figures refer, was called "rateable value"). As a table on paper, its first few lines would look like this:[32]

	population	acreage	rateable value (£)
Bedfordshire	427,970	305,089	12,789,675
Berkshire	585,450	463,830	19,770,843
Buckinghamshire	542,020	477,750	29,709,579

Figure 2

Let's suppose we have these data in an electronic file `countyDataPlus.txt`, in a format having one line per county, no commas in the numbers, and fields separated by the "|" symbol, possibly with adjacent whitespace.[33] In other words the first line of the file might look like this:

```
Bedfordshire | 427970|305089| 12789675
```

Then we can read the data into a Perl program using code such as:

(30)

```
1  while ($line = <INFILE>)
2    {
3    chomp($line);
4    $line =~ s/^\s*(\S.*\S)\s*$/$1/;
5    @items = split(/\s*\|\s*/, $line);
6    ($name, $population, $acreage, $rateable_value) = @items;
7    $name = despace($name);
       ⋮
     }
```

Line 30.4 will remove any whitespace from beginning and/or end of the line read in, 30.5 will break it up into an array of strings separated by the vertical bar possibly with adjacent whitespace, 30.6 will assign the respective elements of the array to individual scalar variables, and 30.7 will modify multi-word county names using the `despace()` subroutine, (19), to replace internal whitespace by single underline characters.

But now we have to decide how to arrange these items into a data-structure within our program, so that later parts of the program can look the information up.

Probably we shall still want to implement the table as an array, rather than a hash with county names as keys – it would not be specially convenient to use a string like `Cambridgeshire_and_Isle_of_Ely` for lookup purposes. But, now we know about hashes, we can choose whether we want the table to be an array of arrays, or an array of hashes. If it is an array of arrays, then the rows are indexed by integers 0 to 45 and the columns by integers 0 to 3:

```
print "$counties[1][0]\n";
```

```
Berkshire
```

```
print "$counties[1][2]\n";
```

```
463830
```

If on the other hand we go for an array of hashes, each row of the array will be a little four-element hash, which could be indexed with the keys `name`, `pop`, `acreage`, `rValue`:

```
print "$counties[1]{name}\n";
```

Berkshire

```
print "$counties[1]{acreage}\n";
```

463830

(Provided our hash keys are all-letter strings like `name` or `acreage`, we don't need to surround them with quotation marks inside the curly brackets – though if, for instance, we used a two-word string with a space in the middle as a key, this would need quotation marks.)

With only four elements, the lookup efficiency of a hash is irrelevant. But it takes a burden off the programmer if he does not need to remember that acreage is in column 2, and so on. Some Perl programmers would argue that this is really the central advantage of hashes, more significant in practice than the point about efficiency of lookup in large tables.

17.5 Hashes versus references to hashes

The same point we saw earlier about arrays of arrays only *seeming* in Perl to be arrays of multi-item elements applies equally to arrays of hashes (and to hashes of arrays, hashes of hashes, ...). So the code which populates our enlarged `@counties` array with data will need to insert values one cell at a time:

```
$counties[$i]{name} = $name;
$counties[$i]{pop} = $population;
$counties[$i]{acreage} = $acreage;
$counties[$i]{rValue} = $rateable_value;
```

That is, after the `open` statement which makes INFILE the handle for our file of data, the program will run like this:

(31)

```
1   $i = 0;
2   while ($line = <INFILE>)
3     {
4     chomp($line);
5     $line =~ s/^\s*(\S.*\S)\s*$/$1/;
6     @items = split(/\s*\|\s*/, $line);
7     ($name, $population, $acreage, $rateable_value) = @items;
8     $name = despace($name);
9     $counties[$i]{name} = $name;
10    $counties[$i]{pop} = $population;
11    $counties[$i]{acreage} = $acreage;
12    $counties[$i]{rValue} = $rateable_value;
13    ++$i;
14    }
15  close(INFILE);
```

We could replace 31.7–8 with code that turns the items found in the current input line into a four-element hash (using the symbol `=>` to link key/value pairs in a list), like this:

```
%countyHash = (name => despace($name), pop => $population, acreage
   => $acreage, rValue => $rateable_value);
```

but, if we were to do that, we could not go on to replace 31.9–12 by a single statement:

```
%counties[$i] = %countyHash;   #WON'T WORK!
```

Likewise, if we need all the data about a given county, we shall need to extract it from the hash one item at a time:

```
$name = $counties[$i]{name};
$population = $counties[$i]{pop};
   etc.
```

But, provided we respect this constraint, we can forget that appearance is different from reality. (And, once you go on in due course to learn about references, you will be able to escape from the constraint.)