Chapter 6. Hashes

In this chapter, you will see a feature that makes Perl one of the world's truly great programming languages: *hashes*. Although hashes are a powerful and useful feature, you may have used other powerful languages for years without ever hearing of hashes. But you'll use hashes in nearly every Perl program you write from now on; they're that important.

What Is a Hash?

A hash is a data structure, not unlike an array in that it can hold any number of values and retrieve them at will. But instead of indexing the values by *number*, as you did with arrays, you look up hash values by *name*. That is, the indices, called *keys*, aren't numbers but instead are arbitrary, unique strings (see Figure 6-1).

Hash keys are strings, first of all, so instead of getting element number `3` from an array, you access the hash element named `wilma`, for instance.

These keys are arbitrary strings—you can use any string expression for a hash key. And they are unique strings—just as there's only one array element numbered `3`, there's only one hash element named `wilma`.

Another way to think of a hash is that it's like a barrel of data, where each piece of data has a tag attached. You can reach into the barrel and pull out any tag and see what piece of data is attached. But there's no "first" item in the barrel; it's just a jumble. In an array, you start with element `0`, then element `1`, then element `2`, and so on. But in a hash there's no fixed order, no first element. It's just a collection of key-value pairs.

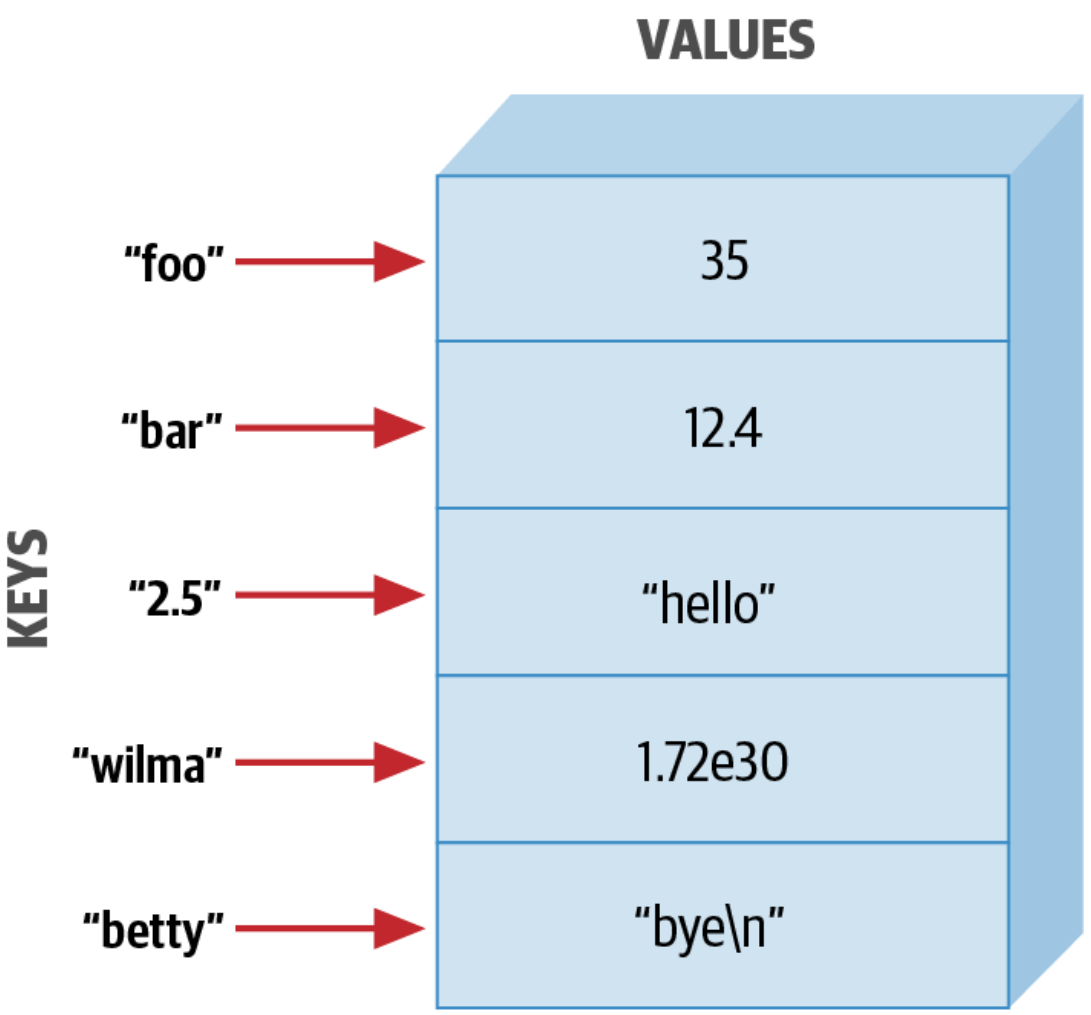

Figure 6-1. Hash keys and values

The keys and values are both arbitrary scalars, but the keys are always converted to strings. So, if you used the numeric expression `50/20` as the key, it would be turned into the three-character string `"2.5"`, which is one of the keys shown in Figure 6-2.

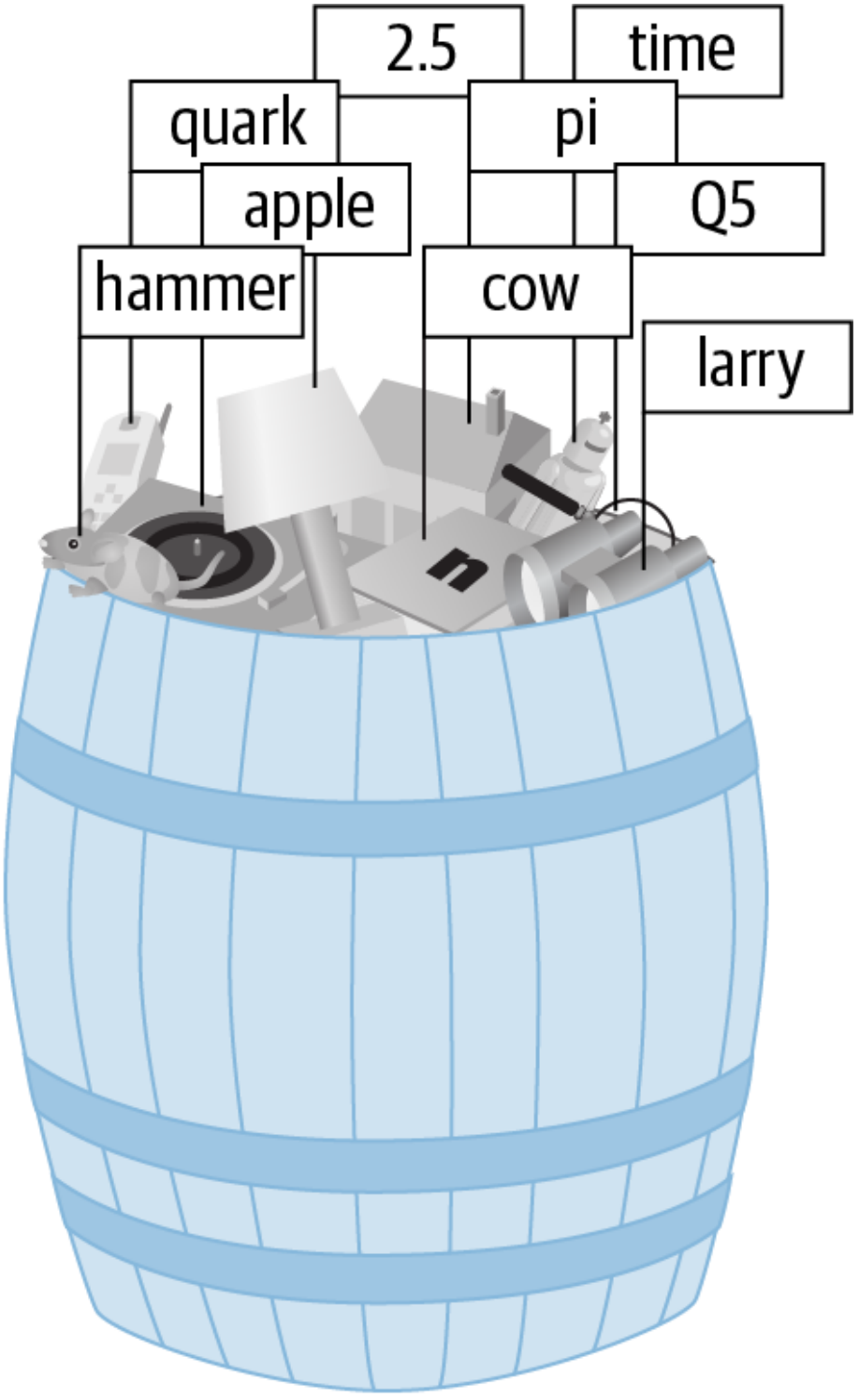

Figure 6-2. A hash as a barrel of data

As usual, Perl's "no unnecessary limits" philosophy applies: a hash may be of any size, from an empty hash with zero key-value pairs, up to whatever fills up your memory.

Some implementations of hashes (such as in the original *awk* language, where Larry borrowed the idea from) slow down as the hashes get larger and larger. This is not the case in Perl—it has a good, efficient, scalable algorithm. So, if a hash has only three key-value pairs, it's very quick to "reach into the barrel" and pull out any one of those. If the hash has three

million key-value pairs, it should be just about as quick to pull out any one of those. A big hash is nothing to fear.

It's worth mentioning again that the keys are always unique, although you may use the same value more than once. The values of a hash may be all numbers, all strings, `undef` values, or a mixture. But the keys are all arbitrary, unique strings.

Why Use a Hash?

When you first hear about hashes, especially if you've lived a long and productive life as a programmer using languages that don't have hashes, you may wonder why anyone would want one of these strange beasts. Well, the general idea is that you'll have one set of data "related to" another set of data. For example, here are some hashes you might find in typical applications of Perl:

Driver's license number, name

There may be many, many people named John Smith, but you hope that each one has a different driver's license number. That number makes for a unique key, and the person's name is the value.

Word, count of number of times that word appears

This is a very common use of a hash. It's so common, in fact, that it just might turn up in the exercises at the end of the chapter!

The idea here is that you want to know how often each word appears in a given document. Perhaps you're building an index to a number of documents so that when a user searches for `fred`, you'll know that a certain document mentions `fred` five times, another mentions `fred` seven times, and yet another doesn't mention `fred` at all—so you'll know which documents the user is likely to want. As the index-making program reads through a given document, each time it sees a mention of `fred`, it adds one to the value filed under the key of `fred`. That is, if you had seen `fred` twice already in this document, the value would be `2`, but now you increment it to `3`. If you had not yet seen `fred`, you change the value from `undef` (the implicit, default value) to `1`.

Username, number of disk blocks they are using (wasting)

System administrators like this one: the usernames on a given system are all unique strings, so they can be used as keys in a hash to look up information about that user.

Yet another way to think of a hash is as a *very* simple database, in which just one piece of data may be filed under each key. In fact, if your task description includes phrases like “finding duplicates,” “unique,” “cross-reference,” or “lookup table,” it’s likely that a hash will be useful in the implementation.

Hash Element Access

To access an element of a hash, you use syntax that looks like this:

```
$hash{$some_key}
```

This is similar to what you used for array access, but here you use curly braces instead of square brackets around the subscript (key). You do this because you’re doing something fancier than ordinary array access, so you should use fancier punctuation. And that key expression is now a string rather than a number:

```
$family_name{'fred'}   = 'flintstone';
$family_name{'barney'} = 'rubble';
```

Figure 6-3 shows how the resulting hash keys are assigned.

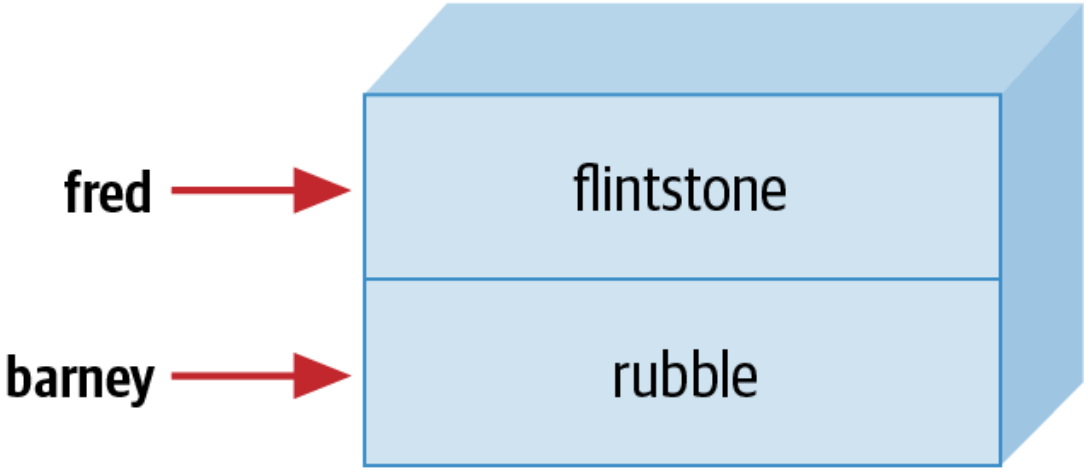

Figure 6-3. Assigned hash keys

This lets you use code like this:

```
foreach my $person (qw< barney fred >) {
  print "I've heard of $person $family_name{$person}.\n";
}
```

The name of the hash is like any other Perl identifier. And it's from a separate namespace; that is, there is no connection between the hash element `$family_name{"fred"}` and a subroutine `&family_name`, for example. Of course, there's no reason to confuse everyone by giving everything the same name. But Perl won't mind if you also have a scalar called `$family_name` and array elements like `$family_name[5]`. We humans will have to do as Perl does; that is, you look to see what punctuation appears before and after the identifier to see what it means. When there is a dollar sign in front of the name and curly braces afterward, you're accessing a hash element.

When choosing the name of a hash, it's often nice to think of the word *for* between the name of the hash and the key, as in "the `family_name` for `fred` is `flintstone`." So the hash is named `family_name`. Then the relationship between the keys and their values becomes clear.

Of course, the hash key may be any expression, not just the literal strings and simple scalar variables that you're showing here:

```
$foo = 'bar';
print $family_name{ $foo . 'ney' };  # prints 'rubble'
```

When you store something in an existing hash element, it overwrites the previous value:

```
$family_name{'fred'} = 'astaire';  # gives new value to existing element
$bedrock = $family_name{'fred'};   # gets 'astaire'; old value is lost
```

That's analogous to what happens with arrays and scalars; if you store something new in `$pebbles[17]` or `$dino`, the old value is replaced. If you store something new in `$family_name{'fred'}`, the old value is replaced as well.

Hash elements spring into existence when you first assign to them:

```
$family_name{'wilma'} = 'flintstone';              # adds a new key (and value)
$family_name{'betty'} .= $family_name{'barney'};  # creates the element if neede
```

This is a feature called *autovivification*, which we talk about more in *Intermediate Perl*. That's also just like what happens with arrays and scalars; if you didn't have `$pebbles[17]` or `$dino` before, you will have it after you assign to it. If you didn't have `$family_name{'betty'}` before, you do now.

And accessing outside the hash gives `undef`:

```
$granite = $family_name{'larry'};  # No larry here: undef
```

Once again, this is just like what happens with arrays and scalars; if there's nothing yet stored in `$pebbles[17]` or `$dino`, accessing them will yield `undef`. If there's nothing yet stored in `$family_name{'larry'}`, accessing it will yield `undef`.

The Hash as a Whole

To refer to the entire hash, use the percent sign (`%`) as a prefix. So, the hash you've been using for the last few pages is actually called `%family_name`.

For convenience, you can convert a hash into a list and back again. Assigning to a hash (in this case, the one from Figure 6-1) is a list-context assignment, where the list is key-value pairs:

```
%some_hash = ('foo', 35, 'bar', 12.4, 2.5, 'hello',
      'wilma', 1.72e30, 'betty', "bye\n");
```

Although you can use any list expression, it must have an even number of elements, because the hash is made of key-value *pairs*. An odd element will likely do something unreliable, although it's a warnable offense.

The value of the hash (in a list context) is a simple list of key-value pairs:

```
my @any_array = %some_hash;
```

Perl calls this *unwinding* the hash—turning it back into a list of key-value pairs. Of course, the pairs won't necessarily be in the same order as the original list:

```
print "@any_array\n";
  # might give something like this:
  # betty bye (and a newline) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

NOTE

In scalar context, a hash returns the number of keys in the hash: `my $count = %hash;`. However, before v5.26, this returned a weird fraction related to the amount of the hash you have used versus the amount *perl* has allocated.

The order is jumbled because Perl keeps the key-value pairs in an order that's convenient for Perl so that it can look up any item quickly. You use a hash either when you don't care what order the items are in, or when you have an easy way to put them into the order you want.

Of course, even though the order of the key-value pairs is jumbled, each key "sticks" with its corresponding value in the resulting list. So, even

though you don't know where the key `foo` will appear in the list, you know that its value, `35`, will be right after it.

Hash Assignment

It's rare to do so, but you can copy a hash using the obvious syntax of simply assigning one hash to another:

```
my %new_hash = %old_hash;
```

This is actually more work for Perl than meets the eye. Unlike what happens in languages like Pascal or C, where such an operation would be a simple matter of copying a block of memory, Perl's data structures are more complex. So, that line of code tells Perl to unwind the `%old_hash` into a list of key-value pairs, then assign those to `%new_hash`, building it up one key-value pair at a time.

It's more common to transform the hash in some way, though. For example, you could make an inverse hash:

```
my %inverse_hash = reverse %any_hash;
```

This takes `%any_hash` and unwinds it into a list of key-value pairs, making a list like `(` *key, value, key, value, key, value, …* `)`. Then `reverse` turns that list end for end, making a list like `(` *value, key, value, key, value, key, …* `)`. Now the keys are where the values used to be, and the values are where the keys used to be. When you store that in `%inverse_hash`, you can look up a string that was a value in `%any_hash`—it's now a key of `%inverse_hash`. And the value you find is one of the keys from `%any_hash`. So, you have a way to look up a "value" (now a key), and find a "key" (now a value).

Of course, you might guess (or determine from scientific principles, if you're clever) that this will work properly only if the values in the original hash were unique—otherwise, you'd have duplicate keys in the new hash, and keys are always unique. Here's the rule that Perl uses: the last one in, wins. That is, the later items in the list overwrite any earlier ones.

Of course, you don't know what order the key-value pairs will have in this list, so there's no telling which ones would win. You'd use this technique only if you know there are no duplicates among the original values. But that's the case for the IP address and hostname examples given earlier:

```
%ip_address = reverse %host_name;
```

Now you can look up a hostname or IP address with equal ease to find the corresponding IP address or hostname.

The Big Arrow

When assigning a list to a hash, sometimes it's not obvious which elements are keys and which are values. For example, in this assignment (which you saw earlier), we humans have to count through the list, saying "key, value, key, value…" in order to determine whether `2.5` is a key or a value:

```
%some_hash = ('foo', 35, 'bar', 12.4, 2.5, 'hello',
      'wilma', 1.72e30, 'betty', "bye\n");
```

Wouldn't it be nice if Perl gave you a way to pair up keys and values in that kind of a list so that it would be easy to see which ones were which? Larry thought so, which is why he invented the big arrow (`=>`). To Perl, it's just a different way to "spell" a comma, so it's also sometimes called the "fat comma." That is, in the Perl grammar, any time you need a comma (`,`), you can use the big arrow instead; it's all the same to Perl. So here's another way to set up the hash of last names:

```
my %last_name = (  # a hash may be a lexical variable
  'fred'   => 'flintstone',
  'dino'   => undef,
  'barney' => 'rubble',
  'betty'  => 'rubble',
);
```

Here, it's easy (or perhaps easier, at least) to see whose name pairs with which value, even if we end up putting many pairs on one line. And notice that there's an extra comma at the end of the list. As we saw earlier, this is harmless but convenient; if we need to add additional people to this hash, we'll simply make sure that each line has a key-value pair and a trailing comma. Perl will see that there is a comma between each item and the next, and one extra (harmless) comma at the end of the list.

It gets better, though. Perl offers many shortcuts that can help the programmer. Here's a handy one: you may omit the quote marks on some hash keys when you use the fat comma, which automatically quotes the values to its left:

```
my %last_name = (
  fred   => 'flintstone',
  dino   => undef,
  barney => 'rubble',
  betty  => 'rubble',
);
```

Of course, you can't omit the quote marks on just *any* key, since a hash key may be any arbitrary string. If that value on the left looks like a Perl operator, Perl can get confused. This won't work because Perl thinks the `+` is the addition operator, not a string to quote:

```
my %last_name = (
  +   => 'flintstone',  # WRONG! Compilation error!
);
```

But keys are often simple. If the hash key is made up of nothing but letters, digits, and underscores (without starting with a digit), you *may* be able to omit the quote marks. This kind of simple string without quote marks is called a *bareword*, since it stands alone without quotes.

Another place you are permitted to use this shortcut is the most common place a hash key appears: in the curly braces of a hash element reference. For example, instead of `$score{'fred'}`, you could write simply `$score{fred}`. Since many hash keys are simple like this, not using quotes is a real convenience. But beware; if there's anything inside the

curly braces besides a bareword, Perl will interpret it as an expression. For instance, if there is a `.`, Perl interprets it as a string concatenation:

```
$hash{ bar.foo } = 1;  # that's the key 'barfoo'
```

Hash Functions

Naturally, there are some useful functions that can work on an entire hash at once.

The keys and values Functions

The `keys` function yields a list of all the keys in a hash, while the `values` function gives the corresponding values. If there are no elements to the hash, then either function returns an empty list:

```
my %hash = ('a' => 1, 'b' => 2, 'c' => 3);
my @k = keys %hash;
my @v = values %hash;
```

So, `@k` will contain `'a'`, `'b'`, and `'c'`, and `@v` will contain `1`, `2`, and `3`—in *some* order. Remember, Perl doesn't maintain the order of elements in a hash. But whatever order the keys are in, the values are in the corresponding order: if `'b'` is last in the keys, `2` will be last in the values; if `'c'` is the first key, `3` will be the first value. That's true as long as you don't modify the hash between the request for the keys and the one for the values. If you add elements to the hash, Perl reserves the right to rearrange it as needed, to keep the access quick. In a scalar context, these functions give the number of elements (key-value pairs) in the hash. They do this quite efficiently, without having to visit each element of the hash:

```
my $count = keys %hash;  # gets 3, meaning three key-value pairs
```

Once in a long while, you'll see that someone has used a hash as a Boolean (true/false) expression, something like this:

```
if (%hash) {
  print "That was a true value!\n";
}
```

That will be true if (and only if) the hash has at least one key-value pair. So, it's just saying, "If the hash is not empty…." But this is a pretty rare construct, as such things go. The actual result is the number of keys (v5.26 and later) or the internal debugging string useful to the people who maintain Perl (prior to v5.26). It used to look something like "4/16," but either version of the value is guaranteed to be true when the hash is nonempty and false when it's empty, so the rest of us can still use it for that.

The each Function

If you wish to iterate over (that is, examine every element of) an entire hash, one of the usual ways is to use the `each` function, which returns a key-value pair as a two-element list. On each evaluation of this function for the same hash, the next successive key-value pair is returned, until you have accessed all the elements. When there are no more pairs, `each` returns an empty list.

In practice, the only way to use `each` is in a `while` loop, something like this:

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

There's a lot going on here. First, `each %hash` returns a key-value pair from the hash, as a two-element list; let's say the key is `"c"` and the value is `3`, so the list is `("c", 3)`. That list is assigned to the list `($key, $value)`, so `$key` becomes `"c"` and `$value` becomes `3`.

But that list assignment is happening in the conditional expression of the `while` loop, which is a scalar context. (Specifically, it's a Boolean context, looking for a true/false value; and a Boolean context is a particular kind of scalar context.) The value of a list assignment in a scalar context is the

number of elements in the source list— `2`, in this case. Since `2` is a true value, you enter the body of the loop and print the message `c => 3`.

The next time through the loop, `each %hash` gives a new key-value pair; say it's `("a", 1)` this time. (It knows to return a different pair than previously because it keeps track of where it is; in technical jargon, there's an iterator stored in each hash.) Those two items are stored in `($key, $value)`. Since the number of elements in the source list was again `2`, a true value, the `while` condition is true, and the loop body runs again, telling us `a => 1`.

NOTE

Each hash has its own private iterator. Loops using `each` may be nested as long as they are iterating over *different* hashes. Different calls to `each` on the same hash will probably give you unexpected results since they interfere with each other.

You go one more time through the loop, and by now you know what to expect, so it's no surprise to see `b => 2` appear in the output.

But you knew it couldn't go on forever. Now, when Perl evaluates `each %hash`, there are no more key-value pairs available, so `each` has to return an empty list. The empty list is assigned to `($key, $value)`, so `$key` gets `undef` and `$value` also gets `undef`.

But that hardly matters, because you're evaluating the whole thing in the conditional expression of the `while` loop. The value of a list assignment in a scalar context is the number of elements in the source list—in this case, that's `0`. Since `0` is a false value, the `while` loop is done, and execution continues with the rest of the program.

Of course, `each` returns the key-value pairs in a jumbled order. (It's the same order as `keys` and `values` would give, incidentally—the "natural" order of the hash.) If you need to go through the hash in order, simply sort the keys, perhaps something like this:

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
  print "$key => $value\n";
  # Or, we could have avoided the extra $value variable:
  #  print "$key => $hash{$key}\n";
}
```

We'll see more about sorting hashes in Chapter 14.

Typical Use of a Hash

At this point, you may find it helpful to see a more concrete example.

The Bedrock Library uses a Perl program in which a hash keeps track of how many books each person has checked out, among other information:

```
$books{'fred'} = 3;
$books{'wilma'} = 1;
```

It's easy to see whether an element of the hash is true or false; do this:

```
if ($books{$someone}) {
  print "$someone has at least one book checked out.\n";
}
```

But there are some elements of the hash that aren't true:

```
$books{"barney"}  = 0;      # no books currently checked out
$books{"pebbles"} = undef;  # no books EVER checked out; a new library card
```

Since Pebbles has never checked out any books, her entry has the value of `undef` rather than `0`.

There's a key in the hash for everyone who has a library card. For each key (that is, for each library patron), there's a value that is either a number of books checked out, or `undef` if that person's library card has never been used.

The exists Function

To see whether a key exists in the hash (that is, whether someone has a library card or not), use the `exists` function, which returns a true value if the given key exists in the hash, whether the corresponding value is true or not:

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
}
```

That is to say, `exists $books{"dino"}` will return a true value if (and only if) `dino` is found in the list of keys from `keys %books`.

The delete Function

The `delete` function removes the given key (and its corresponding value) from the hash (if there's no such key, its work is done; there's no warning or error in that case):

```
my $person = "betty";
delete $books{$person};  # Revoke the library card for $person
```

Note that this is *not* the same as storing `undef` in that hash element—in fact, it's precisely the opposite! Checking `exists($books{"betty"})` will give opposite results in these two cases; after a `delete`, the key *can't* exist in the hash, but after storing `undef`, the key *must* exist.

In the example, `delete` versus storing `undef` is the difference between taking away Betty's library card versus giving her a card that has never been used.

Hash Element Interpolation

You can interpolate a single hash element into a double-quoted string just as you'd expect:

```
foreach $person (sort keys %books) {              # each patron, in order
  if ($books{$person}) {
    print "$person has $books{$person} items\n";   # fred has 3 items
  }
}
```

But there's no support for entire hash interpolation; `"%books"` is just the six characters of (literally) `%books`. So you've seen all of the magical characters that need backslashing in double quotes: `$` and `@`, because they introduce a variable that Perl will try to interpolate; `"`, since that's the quoting character that would otherwise end the double-quoted string; and `\`, the backslash itself. Any other characters in a double-quoted string are nonmagical and should simply stand for themselves. But do beware of the apostrophe (`'`), left square bracket (`[`), left curly brace (`{`), small arrow (`->`), or double colon (`::`) following a variable name in a double-quoted string, as they could perhaps mean something you didn't intend.

The %ENV Hash

There's a hash that you can use right away. Your Perl program, like any other program, runs in a certain *environment*, and your program can look at the environment to get information about its surroundings. Perl stores this information in the `%ENV` hash. For instance, you'll probably see a `PATH` key in `%ENV`:

```
print "PATH is $ENV{PATH}\n";
```

Depending on your particular setup and operating system, you'll see something like this:

```
PATH is /usr/local/bin:/usr/bin:/sbin:/usr/sbin
```

Most of these are set for you automatically, but you can add to the environment yourself. How you do this depends on your operating system and shell. For Bash, you'd use `export`:

```
$ export CHARACTER=Fred
```

For Windows, you'd use `set`:

```
C:\> set CHARACTER=Fred
```

Once you set these environment variables outside your Perl program, you can access them inside your Perl program:

```
print "CHARACTER is $ENV{CHARACTER}\n";
```

You'll see more about `%ENV` in Chapter 15.

Exercises

See "Answers to Chapter 6 Exercises" for answers to these exercises:

1. [7] Write a program that will ask the user for a given name and report the corresponding family name. Use the names of people you know, or (if you spend so much time on the computer that you don't know any actual people) use the following table:

Input	Output
`fred`	`flintstone`
`barney`	`rubble`
`wilma`	`flintstone`

2. [15] Write a program that reads a series of words (with one word per line) until end-of-input, then prints a summary of how many times each word was seen. (Hint: remember that when an undefined value is used as if it were a number, Perl automatically converts it to `0`. It may help to look back at the earlier exercise that kept a running total.) So, if the input words were `fred`, `barney`, `fred`, `dino`, `wilma`, `fred`

(all on separate lines), the output should tell us that `fred` was seen `3` times. For extra credit, sort the summary words in code point order in the output.

3. [15] Write a program to list all of the keys and values in `%ENV`. Print the results in two columns in ASCIIbetical order. For extra credit, arrange the output to vertically align both columns. The `length` function can help you figure out how wide to make the first column. Once you get the program running, try setting some new environment variables and ensuring that they show up in your output.