3

Lists and Hashes

As we saw from the previous chapter, there are three types of data: scalars, lists, and hashes. So far we've only been working with scalars – single numbers or strings. We've joined two single strings together to make one, converted one currency only, and held one number in a variable.

There are times, when we'll want to group together information or express correspondences between information. Just like the ingredients in a recipe or the pieces in a jigsaw, some things belong together in a natural sequence, for example, individual lines in a file, or the names of players in a squash ladder. In Perl, we represent these relationships in lists – series of scalars. They can be stored in another type of variable called an **array**, and we call each piece of data in the list an **element**.

Alternatively, some things are better expressed as a set of one-to-one correspondences. A phone book, for example, is a set of correspondences between addresses and phone numbers. In Perl, structures like the phone book are represented as a **hash**. Some people call them 'associative arrays' because they look a bit like arrays where each element is associated with another value. Most Perl programmers find that a bit too long-winded and just call them hashes.

In this chapter, we'll see how we build up lists and hashes and what we can do with them when we've got them. We'll also begin to look at some control structures, which will enable us to step through lists and arrays. As well as all this, we'll learn how to process data more than once without having to write out the relevant sections of our program again and again.

Lists

We're all familiar with lists from everyday life. Think about a shopping list, what properties does it have? First of all, it's a single thing, one piece of paper. Secondly, it's made up of a number of values. In the case of a shopping list, you might want to say that these values are actually strings – `"ketchup"`, `"peanut butter"`, `"ice cream"`, and so on. Finally, it's also ordered, which means that there's a first item and a last item.

Lists in Perl aren't actually that much different: They're counted as a single thing, but they're made up of a number of values. In Perl, these values are scalars, rather than purely strings. They're also stored in the order they were created.

We'll specify lists in our program code as literals, just like we did with strings and numbers. We'll also be able to perform certain operations on them. Let's begin by looking at a few simple lists and how we create them.

Simple Lists

The simplest shopping list is one where you have nothing to buy. Similarly, the simplest list in Perl has no elements in it. Here's what it looks like:

```
()
```

A simple pair of parentheses – that's how we denote a list. However, it's not very interesting. Let's try putting in some values:

```
(42)
("cheese")
```

As you can see, we have created two lists, one containing a number, and one containing a string – so far so good. Now, remember that I said `print` was a list operator? The magic about operators like `print` is that you can omit the brackets. Saying `print "cheese"` is just the same as saying `print("cheese")`. So what we give to `print` is really a list. We're allowed to leave out the parentheses if we wish.

From this, we should be able to work out how to put multiple values into a list. When we said:

```
print("Hello, ", "world", "\n");
```

we were actually passing the following list to the print operator:

```
("Hello ", "world", "\n")
```

As you can see, this is a three-element list, and the elements are separated with commas. Computers and computer people start counting from zero, so here's your chance to practise. The zeroth element is `"Hello "`, the first is `"world"`, and the second is `"\n"`. Now, let's do that again with numbers instead of strings:

```
(123, 456, 789)
```

This is exactly the same as before, and if we were to print this new list, this is what would happen:

```
#!/usr/bin/perl
# numberlist.plx
use warnings;
use strict;

print (123, 456, 789);
```

```
>perl numberlist.plx
123456789>
```

As before, perl doesn't automatically put spaces between list elements for us when it prints them out, it just prints them as it sees them. Similarly, it doesn't put a new line on the end for us. If we want to add spaces and new lines, then we need to put them into the list ourselves.

More Complex Lists

We can also mix strings, numbers, and variables in our lists. Let's see an example of a list with several different types of data in it:

Try It Out – Mixed Lists

Although this isn't very different from what we were doing with print in the last chapter, this example reinforces the point that lists can contain any scalar literals and scalar variables. So, type this in, and save it as `mixedlist.plx`.

```
#!/usr/bin/perl
# mixedlist.plx
use warnings;
use strict;

my $test = 30;
print
     "Here is a list containing strings, (this one) ",
     "numbers (",
     3.6,
     ") and variables: ",
     $test,
     "\n"
;
```

When you run that, here's what you should see:

```
> perl mixedlist.plx
Here is a list containing strings, (this one) numbers (3.6) and variables: 30
>
```

How It Works

This is how we're going to start programs from now on, in order to make sure that we have both warnings and extra checks turned on. Remember that if you're using a version of Perl less than 5.6 you'll need to say `#!/usr/bin/perl -w` for the first line to turn on warnings, and also leave out the `use warnings;` line:

```
#!/usr/bin/perl
# mixedlist.plx
use warnings;
use strict;
```

Next, we initialize our variable. Note that we can declare the variable and give it a value on the same statement. It's exactly the same as doing this:

```
my $test = 30;
```

but is just as clear and saves a line, so it's a common thing to do – it's one of Perl's many idioms:

```
my $test;
$test = 30;
```

Perl is more like a human language than most programming languages. Perl was designed to be easy for humans to write, not for computers to read. Just like human languages, Perl has shortcuts and idioms. Perl programmers do tend to use a lot of these idioms in their code, and you may come across them if you're reading other people's programs. As a result of this, we're not going to shy away from those idioms, even if they can be slightly confusing at times. Instead, we'll try taking them apart to see how they work.

Finally, we have our list. It's a list of six elements, including literal strings, literal numbers, and a scalar variable for good measure:

```
print
     "Here is a list containing strings, (this one) ",
     "numbers (",
     3.6,
     ") and variables: ",
     $test,
     "\n"
;
```

Since variables interpolate in double-quoted strings inside lists just as well as at any other time, we could have done that all as one long single-element list:

```
print ("Here is a list containing strings, (this one) numbers (3.6) and variables:
$test\n");
```

There is a disadvantage of writing your code this way. New lines in your string literals will turn into new lines in your output. So, if you keep the maximum length of the lines in your source code to about 80 columns (it's a good idea to keep your programs readable), one long string will wrap over, and you'll see this sort of thing:

```
> perl mixedlist.plx
Here is a list containing strings, (this one) numbers (3.6) and
variables: 30
>
```

So if you're ever printing long strings, consider splitting it up into a list of smaller strings on separate lines as we've done above.

In the same way, single-quoted strings act no differently when they're list elements: `('A number:', '$test')` will actually give you two strings, and if you print out that list, you will see this:

```
A number:$test
```

Similarly, `q//` and `qq//` can be used to delimit strings when you're using them as list elements. There's absolutely no difference between the previous example and `(q/A number:/, q/$test/)`

However, there's another trick. When your lists are made up purely from single words, you can specify them with the `qw//` operator. Just like the `q//` and `qq//` operators, you can choose any paired brackets or non-word characters as your delimiters. The following lists are all identical:

```
('one', 'two', 'three', 'four')
qw/one two three four/
qw(one two three four)
qw<one two three four>
qw{one two three four}
qw[one two three four]
qw|one two three four|
```

You shouldn't separate your words with commas inside `qw//`. In fact, if you do, perl will complain, especially since we always have warnings turned on! For example, if we ran this:

```
#!/usr/bin/perl
# badlist.plx
use warnings;
use strict;
print qw(one,two,three,four);
```

we would quickly see

```
> perl badlist.plx
Possible attempt to separate words with commas at badlist.plx line 5.
Possible attempt to separate words with commas at badlist.plx line 5.
Possible attempt to separate words with commas at badlist.plx line 5.
one,two,three,four>
```

You can use any white space, tabs, or new lines to separate your elements. The same list as above `('one', 'two', 'three', 'four')` can also be written like this:

```
qw(
     one
     two
     three
     four
)
```

One last thing to note is that perl automatically **flattens** lists. That is, if you try putting a list inside another list, the internal list loses its identity. In effect, perl removes all the brackets apart from the outermost pair. There's no difference at all between any of these three lists:

```
(3, 8, 5, 15)
((3, 8), (5, 15))
(3, (8, 5), 15)
```

Similarly, perl sees each of these lists as exactly the same as the others:

```
('one', 'two', 'three', 'four')
(('one', 'two', 'three', 'four'))
(qw(one two three), 'four')
(qw(one two), q(three), 'four')
(qw(one two three four))
```

This doesn't mean that you can't store a list inside another list and keep the structure of the first list intact. For the moment we can't do it, but we'll see how it's done when we look at references in Chapter 7.

Accessing List Values

We've now seen most of the ways of building up lists in Perl, and we can throw lists at list operators like `print`. But another thing we need to be able to do with lists is access a specific element or set of elements within it. The way to do this is to place the number of the elements we want in square brackets after the list, like this:

```
#!/usr/bin/perl
# access.plx
use warnings;
use strict;

print (('salt', 'vinegar', 'mustard', 'pepper')[2]);
print "\n";
```

Before you run this, though, see if you can work out which word will be printed.

```
>perl access.plx
mustard
>
```

Did you think it was going to be 'vinegar'? Don't forget that computers start counting things from zero!

You should also notice that we had to put brackets around the whole thing. This is because the precedence of `print` is extremely high. Without the brackets, perl groups the statement in two parts like this:

```
print('salt', 'vinegar', 'mustard', 'pepper')    [2];
```

This means the whole of the list is passed to `print`, after which perl attempts to retrieve the second element of `print`. The problem is, you can only take an element from a list, and as we already know, `print` isn't a list.

So, since print needs to be passed a list, we make a list out of the element we want:

```
print (
     ('salt', 'vinegar', 'mustard', 'pepper')[2]
);
```

The element you want doesn't have to be given as a literal – variables work just as well. Here's an example we'll draw on later:

Try It Out – Months Of The Year

We'll create a list of the months of the year, and then use a variable to access them. Save this file as `months.plx`:

```
#!/usr/bin/perl
# months.plx
use warnings;
use strict;

my $month = 3;
print qw(
     January    February   March
     April      May        June
     July       August     September
     October    November   December
)[$month];
```

When this is run, you should now be expecting it to give you 'April', and it does:

```
>perl months.plx
April>
```

How It Works

The key piece of code for this example is the last statement:

```
print qw(
     January    February   March
     April      May        June
     July       August     September
     October    November   December
)[$month];
```

We have `$month` as `3`, and so we are telling perl to print out the third element of the list, starting from zero. Because we're using `qw//` we can use arbitrary whitespace, tabs, and new lines to separate each list element, which allows us to present the months in a neat table.

This is exactly the sort of situation that `qw//` was created for. We have a list comprised completely of single words, and we want a way to represent that to perl in a tidy way in our source code. It's far easier to read than spelling the list out longhand, even though these statements are equivalent:

```
print (('January','February', 'March', 'April', 'May', 'June', 'July', 'August',
'September', 'October', 'November', 'December')[$month]);
```

What do you think would happen if we chose a non-integer value for our element? Let's use a value with a fractional part. Change the above file so that line 5 reads:

```
my $month = 2.2;
```

Perl will round the number in this case, and you should get the answer March. In fact, perl always rounds towards zero, so anything between 2 and 3 will get you March.

What about negative numbers? Actually, something interesting happens here – perl starts counting backwards from the end of the list. So element -1 is the last one, -2 the second before last, and so on.

```
#!/usr/bin/perl
# backwards.plx
use warnings;
use strict;

print qw(
     January     February    March
     April       May         June
     July        August      September
     October     November    December
)[-1];
```

And, true to form, we'll get the last element of the array when we run the program.

```
>perl backwards.plx
December>
```

List Slices

So much for getting a single element out of a list. What if we want to get several? Well, instead of putting a number or a scalar variable inside those square brackets, you can actually put a list there instead. For example, this:

```
(19, 68, 47, 60, 53, 51, 58, 55, 47)[(4, 5, 6)]
```

returns another list consisting of elements four, five, and six: `(53, 51, 58)`. Actually, inside the square brackets, we don't need the additional set of parentheses, so you might as well say:

```
(19, 68, 47, 60, 53, 51, 58, 55, 47)[4, 5, 6]
```

We call this getting a **list slice,** and the same methods work with lists of strings:

Try It Out – Multiple Elements Of A List

This program is called `multilist.plx`. Just like the above examples, we're taking several elements from a list:

```
#!/usr/bin/perl
# multilist.plx
use warnings;
use strict;

my $mone; my $mtwo;
($mone, $mtwo) = (1, 3);

print (("heads ", "shoulders ", "knees ", "toes ")[$mone, $mtwo]);
print "\n";
```

Try and think what it's going to produce before you run it. Here's what happens:

```
> perl multilist.plx
shoulders toes
>
```

As you may have realized, we simply printed out the first and the third elements from the list, if you start counting from zero.

How It Works

There are two key tricks in this example. The first is on line seven:

```
    ($mone, $mtwo) = (1, 3);
```

You might be able to see what this line does, from how the rest of the program runs. The value of `$mone` is set to 1, and `$mtwo` to 3. But how does this work?

Perl allows lists on the left-hand side of an assignment operator – we say that lists are legal **lvalues**. When we assign one list to another, the right-hand list is built up first. Then perl assigns each element in turn, from the right hand side of the statement to the left. So 1 is assigned to `$mone`, and then 3 is assigned to `$mtwo`.

If you're okay with that, then now's a good time for a quick quiz. Suppose we've done the above: `$mone` is 1 and `$mtwo` is 3. What do you think would happen if we said this:

```
    ($mone, $mtwo) = ($mtwo, $mone);
```

Well, the right-hand list is built up first, so perl looks at the values of the variables and constructs the list `(3, 1)`. Then the 3 is assigned to `$mone`, and the 1 assigned to `$mtwo`. In effect, we've swapped the values of the variables around – a handy trick to learn and remember. Chances are that it's something you'll need to do again and again over time.

Back to our example! Once we've set `$mone` to 1 and `$mtwo` to 3, we can pick out these elements from a list. There's nothing that says that we have to use literals to pick out the elements we want. This:

```
    print (("heads ", "shoulders ", "knees ", "toes ")[$mone, $mtwo]);
```

is interpreted by perl just the same as this:

```
    print (("heads ", "shoulders ", "knees ", "toes ")[1, 3]);
```

Indeed, both statements equate to the same thing – picking out a list consisting of the first and third elements of our original list and printing them. In effect, we call:

```
    print ("shoulders ", "toes ");
```

which is indeed what happens.

Ranges

Often our lists will be a lot simpler than a group of different values. We'll want to talk about "the numbers 1 to 10" or "the letters a to z." Rather than write them out longhand, Perl gives us the ability to specify a range of numbers or letters. Suppose we say:

```
(1 .. 6)
```

This will give us a list of 6 elements from 1 to 6, exactly the same as if we had said `(1, 2, 3, 4, 5, 6)`. This can really save time when you're dealing with a few hundred elements, but note that this only works for integers. If you'll recall our efforts to use lists to get at elements of another list, the fractional values in the list were rounded towards zero. Exactly the same thing happens here:

```
(1.4 .. 6.9)
```

would produce `(1, 2, 3, 4, 5, 6)` again. There's no problems with using negative numbers in you ranges, though. For example:

```
(-6 .. 3)
```

produces the list `(-6, -5, -4, -3, -2, -1, 0, 1, 2, 3)`

The right-hand number must, however, be higher than the left-hand one, so we can't use this technique to count down. Instead, you can reverse any list using the `reverse` operator, as we'll see very shortly.

We can do the same for letters as well:

```
('a'..'k')
```

This will give us an 11-element list, consisting of each letter from 'a' to 'k' inclusive. Note that we can't mix letters and numbers within a range. If we try, perl will interpret the string as a number, and treat it as zero:

Try It Out – Counting Up And Down

Here's a demonstration of all the things we can do with ranges:

```
#!/usr/bin/perl
# ranges.plx
use warnings;
use strict;

print "Counting up: ", (1 .. 6), "\n";
print "Counting down: ", (6 .. 1), "\n";
print "Counting down  (properly this time) : ", reverse(1 .. 6), "\n";

print "Half the alphabet: ", ('a' .. 'm'), "\n";
print "The other half (backwards): ", reverse('n' .. 'z'), "\n";

print "Going from 3 to z: ", (3 .. 'z'), "\n";
print "Going from z to 3: ", ('z' .. 3), "\n";
```

Which of those will work and which won't? Let's find out…:

```
> perl ranges.plx
Argument "z" isn't numeric in range (or flop) at ranges.plx line 13.
Argument "z" isn't numeric in range (or flop) at ranges.plx line 14.
Counting up: 123456
Counting down:
Counting down  (properly this time): 654321
Half the alphabet: abcdefghijklm
The other half (backwards): zyxwvutsrqpon
Going from 3 to z
Going from z to 30123
>
```

How It Works

After the usual opening, we first count upwards with a range:

```
print "Counting up: ", (1 .. 6), "\n";
```

We've seen the range in action before, and we know this produces `(1, 2, 3, 4, 5, 6)`. We pass `print` a list containing the string `"Counting up: "`, the six elements, and a new line. Because a list inside a list gets flattened, we're actually just passing an eight-element list. It's the same as if we'd done:

```
print "Counting up: ", 1, 2, 3, 4, 5, 6, "\n";
```

And we get the expected result:

```
Counting up: 123456
```

Next, we try and count down:

```
print "Counting down: ", (6 .. 1), "\n";
```

This doesn't work because the right hand side needs to be bigger than the left, and all that's produced is the empty list, `()`. To count down properly, we need to make a list using `(1 .. 6)` as before and turn it around. The `reverse` operator turns any list on its head. For example:

```
reverse (qw(The cat sat on the mat))
```

produces the same as:

```
qw(mat the on sat cat The)
```

In this case, `reverse(1..6)` produces `(1, 2, 3, 4, 5, 6)` and then turns it around to become `(6, 5, 4, 3, 2, 1)`, and we see the list appear in that order:

```
Counting down  (properly this time) : 654321
```

Next we demonstrate a simple alphabetic range:

```
print "Half the alphabet: ", ('a' .. 'm'), "\n";
```

This range expands to the values 'a', 'b', 'c', and so, on all the way to 'm'. Doing that backwards is easy:

```
print "The other half (backwards): ", reverse('n' .. 'z'), "\n";
```

Now we come to the ones that don't work, and it's no surprise that perl warns us against them:

```
Argument "z" isn't numeric in range (or flop) at ranges.plx line 13.
Argument "z" isn't numeric in range (or flop) at ranges.plx line 14.
```

The lines in question are:

```
print "Going from 3 to z: ", (3 .. 'z'), "\n";
print "Going from z to 3: ", ('z' .. 3), "\n";
```

What does the error message mean? Well, pretty much what it says: we gave an argument of 'z' to a range, when it was expecting a number instead. The interpreter converted the 'z' to a number, as per the rules in the last chapter, and got a 0. It's equivalent to this:

```
print "Going from 3 to z: ", (3 .. 0), "\n";
print "Going from z to 3: ", (0 .. 3), "\n";
```

The first one produces an empty list, and the second one counts up from 0 to 3.

Combining Ranges and Slices

We can, of course, use ranges in our list slices. The following gets March through September:

```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[2..8])
```

And this gets November through February via December and January (remember that –2 is the second to last, and -1 the last):

```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[-2..1])
```

We can also use a mixture of ranges and literals in our slice. This gives you January, April, and August to December:

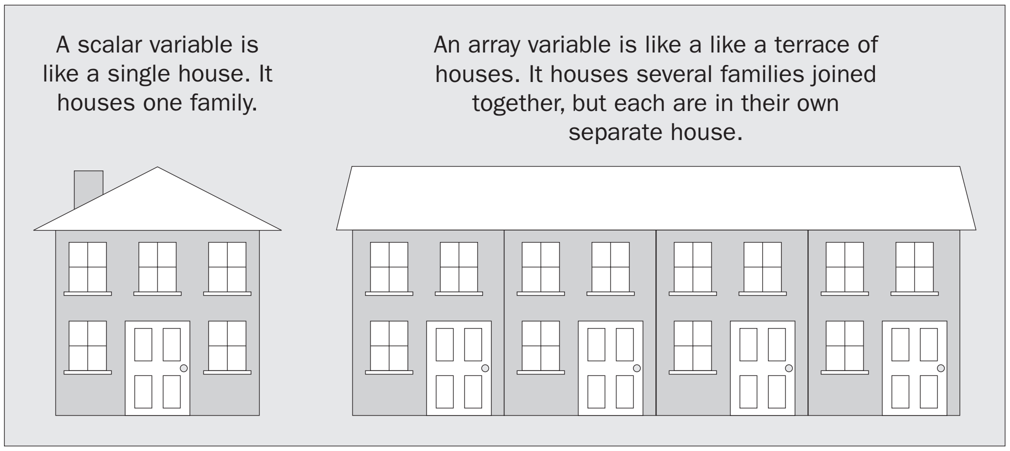
```
(qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec)[0,3,7..11])
```

It may be a bit confusing, but have a go at slicing your own arrays, and you'll get the hang of it in no time at all.

Arrays

Just as with scalars, there's only so much you can do with literals. Literal lists get cumbersome to repeat and don't allow us to manipulate them at all. If we wanted to say 'the same list, but without the last element', we couldn't do it. As before, we need to find a way to store them in a variable.

The variable storage we use for lists is called an **array**. Whereas the name of a scalar variable started with a dollar sign, arrays start with an at sign (@). The same rules for naming your arrays apply as for any other variables: start with an alphabetic character or underscore, followed by one or more alphabetic characters, underscores, or numbers:


A scalar variable is like a single house. It houses one family.
An array variable is like a like a terrace of houses. It houses several families joined together, but each are in their own separate house.

Assigning Arrays

We store a list in an array just like we store a scalar literal into a scalar variable, by assigning it with =:

```
@array = (1,2,3);
```

Once we've assigned our array, we can use our array where we would use a list:

```
#!/usr/bin/perl
# dayarray.plx
use warnings;
use strict;

my @days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
print @days, "\n";
```

This prints:

```
> perl dayarray.plx
MondayTuesdayWednesdayThursdayFridaySaturdaySunday
>
```

Note that `$days` is a completely different variable from `@days` – setting one does nothing to the other. In fact, if you were to do this:

```
#!/usr/bin/perl
# baddayarray1.plx
use warnings;
use strict;

my @days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$days = 31;
```

you would get the following error:

```
Global symbol "$days" requires explicit package name at dayarray.plx line 8.
```

This is because you have declared `@days` to be a lexical variable, but not `$days`. Even when you declare them both, setting one has no effect on the other:

```
#!/usr/bin/perl
# baddayarray2.plx
use warnings;
use strict;

my @days;
my $days;
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$days = 31;
print @days, "\n";
print $days, "\n";
```

prints:

```
MondayTuesdayWednesdayThursdayFridaySaturdaySunday
31
```

What would happen if you assigned an array to a scalar variable? Well, let's see:

Try It Out - Assigning An Array To A Scalar

Here's an example of two arrays that we will assign to two different scalar variables:

```
#!/usr/bin/perl
# arraylen.plx
use warnings;
use strict;

my @array1;
my $scalar1;
@array1 = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
$scalar1 = @array1;

print "Array 1 is @array1\nScalar 1 is $scalar1\n";

my @array2;
my $scalar2;
@array2 = qw(Winter Spring Summer Autumn);
$scalar2 = @array2;

print "Array 2 is @array2\nScalar 2 is $scalar2\n";
```

Save this as `arraylen.plx`, and run it through perl:

```
> perl arraylen.plx
Array 1 is Monday Tuesday Wednesday Thursday Friday Saturday Sunday
Scalar 1 is 7
Array 2 is Winter Spring Summer Autumn
Scalar 2 is 4
>
```

Hmm… The first array has seven elements, and the scalar value is 7. The second has four elements, and the scalar value is 4.

How It Works

Note how array variables interpolate in a double-quoted string. We've seen that if you put a scalar-variable name inside a string, perl will fill in the value of the variable. Now we've put an array variable in a string, and perl has filled it in, but it has placed spaces between the elements. Look at the following two `print` statements:

```
@array = (4, 6, 3, 9, 12, 10);
print @array, "\n";
print "@array\n";
```

The first one does exactly what we've seen with lists, printing all the elements next to each other. The second statement, however, inserts a space between each element:

```
46391210
4 6 3 9 12 10
```

This adding of spaces between elements is what happens when an array is interpolated in a double-quoted string. As with scalars, interpolation is not confined to `print`. For example:

```
$scalar =  "@array\n";
```

is the same as:

```
$scalar = "4 6 3 9 12 10\n";
```

Forcing variables to make sense in a string is called **stringifying** them.

Scalar vs List Context

What happens when we assign an array to a scalar variable? Well, one key point to remember is that perl knows exactly what type of value you want, whether a scalar or an array, at any stage in an operation, and will do its best to make sure you get it.

For example, if we're looking to assign to a scalar variable, we need to have a scalar value – the assignment is taking place in **scalar context**. On the other hand, for example, `print` expects to see a list of arguments. Those arguments are in **list context**. However, some operations may return different values depending on which context they are called. That's what's happening in this case:

```
print @array1;
$scalar1 = @array1;
```

The first line is in list context. In list context, an array returns the list of its elements. In the second line, however, the assignment wants to see a single result, or scalar value, and therefore it is in scalar context. In scalar context, an array returns the number of its elements, in our case, 7 for the days and 4 for the seasons.

If we were to do this:

```
@array2 = @array1;
```

we would be assigning to an array. So we're looking for a **list** of values to fill `@array2`. Here, we're back in list context, and so `@array2` gets filled with all of the values of `@array1`.

We can force something to be in scalar context when it expects to be in list context by using the `scalar` operator. Compare these two statements:

```
print @array1;
print scalar @array1;
```

As we've explained before, `print` usually wants a list, so perl evaluates `print`'s arguments in list context. In the example above, `print` is looking to get a list from each of arguments. That's why the first statement prints the contents of `@array1`. If we force `@array1` into scalar context, then the number of elements in the array is passed to `print` and not the contents of the array:

Perl distinguishes between operations that want a list and operations that want a scalar. Those that want a list, such as `print` or assigning to an array, are said to be in list context. Those that want a scalar are said to be in scalar context. The value of an array in list context is the list of its elements – the value in scalar context is the number of its elements.

Adding to an Array

How do we add elements to an array? Well, one way to do it is by using the 'list flattening' principle and treating our arrays as lists. This isn't a particularly good way to do it, but it works:

```
#!/usr/bin/perl
# addelem.plx
use warnings;
use strict;

my @array1 = (1, 2, 3);
my @array2;
@array2 = (@array1, 4, 5, 6);
print "@array2\n";

@array2 = (3, 5, 7, 9);
@array2 = (1, @array2, 11);
print "@array2\n";
```

```
>perl addelem.plx
1 2 3 4 5 6
1 3 5 7 9 11
>
```

It's far better, however, to use the functions we're going to see later on – `push`, `pop`, `shift`, and `unshift`.

Accessing an Array

Once we've got our list of scalars into an array, it would make sense to be able to get them back out again. We do this slightly differently to the way we get values out of lists.

Accessing Single Elements

So, we can now put elements into an array:

```
my @array = (10, 20, 30);
```

If we look at the array in scalar context, we get the number of elements in it. So:

```
print scalar @array;
```

will print the value 3. But how do we get at one of those elements? We could use the list assignment we were looking at earlier:

```
my $scalar1; my $scalar2; my $scalar3;
($scalar1, $scalar2, $scalar3) = @array;
print "Scalar one is $scalar1\n";
print "Scalar two is $scalar2\n";
print "Scalar three is $scalar3\n";
```

This will print out each of the elements:

```
Scalar one is 10
Scalar two is 20
Scalar three is 30
```

To get at a single element, we do something quite similar to what we did with a list. To get a single element from a list, if you remember, we put the number we want in square brackets after it:

```
$a = (10, 20, 30)[0];
```

This will set `$a` to the zeroth element, 10. We could do this:

```
$a = (@array)[0];
```

in exactly the same way. However, it's more usual to write that as follows:

```
$a = $array[0];
```

Look carefully at that. Even though `@array` and `$array` are different variables, we use the `$array[]` form. Why?

The prime rule is this: the prefix represents what you want to get, not what you've got. So `@` represents a list of values, and `$` represents a single scalar. Hence, when we're getting a single scalar from an array, we never prefix the variable with `@` – that would mean a list. A single scalar is always prefixed with a `$`.

`$array[0]` and `@array` aren't related – `$array[0]` can only refer to an element of the `@array` array. If you try and use the wrong prefix, perl will complain with a warning:

```
#!/usr/bin/perl
# badprefix.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
print @array[1];
```

will print:

```
>perl badprefix.plx
Scalar value @array[1] better written as $array[1] at badprefix.plx line 8.
3>
```

We call the number in the square brackets the **array index** or **array subscript**. The array index is the number of the element that we want to get hold of. Back in our little street, we could explain arrays like so:

The collection of houses is @ array; each house is a scalar, and is therefore $array [0]...$array[3]
[0]
[1]
[2]
[3]
@ array Street

Just like extracting elements from lists, we can use a scalar variable as our subscript:

```
#!/usr/bin/perl
# scalarsub.plx
use warnings;
use strict;

my @array = (1, 3, 5, 7, 9);
my $subscript = 3;
print $array[$subscript], "\n";
$array[$subscript] = 12;
```

This prints the third element from zero, which has the value `7`. It then changes that 7 to a 12. Negative subscripts work from the end; as before, `$array[-1]` will give you the last element in the array.

Now let's write something to extract a given element from an array:

Try It Out – The Joke Machine

We'll use arrays to write a program to tell us some (really bad) jokes. We actually set up two arrays – one containing the question, and one containing the answer:

```
#!/usr/bin/perl
# joke1.plx
use warnings;
use strict;

my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);

print "Please enter a number between 1 and 4: ";
my $selection = <STDIN>;
$selection -= 1;
print "How many $questions[$selection] ";
print "programmers does it take to change a lightbulb?\n\n";
sleep 2;
print $punchlines[$selection], "\n";
```

```
> perl joke1.plx
Please enter a number between 1 and 4: 3
How many Perl programmers does it take to change a lightbulb?

A million. One to change it, the rest to try and do it in fewer lines.
```

Hmm. I don't think I'm ready for the move into stand-up comedy quite yet.

How It Works

We first set up our arrays. One is a list of words and so we can use `qw//` to specify it. The other is a list of strings, so we use the ordinary list style:

```
my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);
```

We now ask the user to choose their joke:

```
print "Please enter a number between 1 and 4: ";
my $selection = <STDIN>;
$selection -= 1;
```

Why take one from it? Well, we've asked for a number between one and four, and our array subscripts go from zero to three.

Next we display the set-up line:

```
print "How many $questions[$selection] ";
print "programmers does it take to change a lightbulb?\n\n";
```

From the first line, we see that array elements stringify just like scalar variables. Next, this new function `sleep`:

```
sleep 2;
```

What `sleep` does, as you'll know if you've run the program, is pause the program's operation for a number of seconds. In this case, we're telling it to sleep for two seconds.

After the user has had time to think about it, we display the punchline:

```
print $punchlines[$selection], "\n";
```

Hopefully, you're starting to see alternative ways we can use arrays by now. Of course, we've only been pulling single values from arrays so far. The next logical step is to start working with multiple array elements.

Accessing Multiple Elements

If you'll recall, we created and used a list slice by putting ranges or several numbers in brackets to get multiple elements from a list. If we want to get multiple elements from an array, we can use the analogous concept, an **array slice**.

List slices, if you remember, looked like this:

```
(qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec))[3,5,7..9]
```

Can you work out which elements the slice above consists of? If not, write a short Perl program to print them out, and see if you can get it to separate them with spaces. (Hint: Only arrays stringify with spaces, so you'll need to use one.)

Array slices look very similar. However, now that we are accessing multiple elements and expecting a list, we no longer want to use `$` as the prefix – now we should be using `@`.

We can get the same list as the above like this:

```
my @array = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);
print @array[3,5,7..9];
```

Array slices act like any normal list, and so can be used as an lvalue. Here's a load of slices to mess around with:

Try It Out - Array Slices

Here are a year's sales results for a fictitious bathroom tile shop:

```
#!/usr/bin/perl
# aslice.plx
use warnings;
use strict;

my @sales = (69, 118, 97, 110, 103, 101, 108, 105, 76, 111, 118, 101);
my @months = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);

print "Second quarter sales:\n";
print "@months[3..5]\n@sales[3..5]\n";
my @q2=@sales[3..5];

# Incorrect results in May, August, Oct, Nov and Dec!
@sales[4, 7, 9..11] = (68, 101, 114, 111, 117);

# Swap April and May
@months[3,4] = @months[4,3];
```

Most of the work is behind the scenes, but this is what you'd see if you run it:

```
Second quarter sales:
May Apr Jun
110 103 101
```

Let's take a look at what's actually going on.

How It Works

We set up our two arrays – one holding our sales figures, and the other holding the names of the months:

```
my @sales = (69, 118, 97, 110, 103, 101, 108, 105, 76, 111, 118, 101);
my @months = qw(Jan Feb Mar May Apr Jun Jul Aug Sep Oct Nov Dec);
```

To extract the information about the second quarter, we use an array slice for the months in question:

```
print "Second quarter sales:\n";
print "@months[3..5]\n@sales[3..5]\n";
my @q2=@sales[3..5];
```

In addition to saving the relevant elements to another array, we can also print out the slice and it will be stringified. We can also assign values to an array slice, as well as getting data from it:

```
@sales[4, 7, 9..11] = (68, 101, 114, 111, 117);
```

This sets new values for `$sales[4]`, `$sales[7]`, `$sales[9]`, `$sales[10]` and `$sales[11]`.

Finally, we can use something similar to the `($a, $b) = ($b, $a)` list trick to swap two array elements:

```
@months[3,4] = @months[4,3];
```

This is exactly the same as the following statement:

```
($months[3], $months[4]) = ($months[4], $months[3]);
```

As you can see, this isn't all that far from the list assignment to swap two variables:

```
($mone, $mtwo) = ($mtwo, $mone);
```

Watch your parentheses and square brackets, though.

Running through Arrays

One thing we'll want to do quite often is run over each of the elements in an array or list in turn. If we want to double every value in an array, then **for** each element we come across, we multiply by two. The keyword to use here is `for`. Here's a **for loop**, which prints each element of an array, followed by a new line:

```
#!/usr/bin/perl
# forloop1.plx
use warnings;
use strict;

my @array = qw(America Asia Europe Africa);
my $element;
for $element (@array) {
     print $element, "\n";
}
```

We set up an array, and we declare another scalar variable, `$element`. What we then say is 'set each element of `@array` to `$element` in turn, and then do all the statements in the following block'. So, on our first iteration, `$element` is set to `America`, and then the `print` statement is run. Then `$element` is set to `Asia`, and the `print` statement runs again. This continues until the end of the array is reached.

This should print:

```
>perl forloop1.plx
America
Asia
Europe
Africa
>
```

`$element` is called an **iterator variable** or **loop variable**. It's what we 'see' when we look at each element in turn. This is the syntax of the for loop:

```
for <ITERATOR> (<LIST OR ARRAY>) <BLOCK>
```

The block must start with an opening brace and end with a closing brace, and the list or array that we're running over must be surrounded by parentheses. If we don't supply an iterator variable of our own, perl uses the special `$_` variable, which is often used in Perl functions as a 'default value'. Note that the `for` loop doesn't require a semicolon after the block.

So, when processing a for loop, perl makes the iterator a copy of each element of the list or array in turn, and then runs the block. If the block happens to change the value of the iterator, the corresponding array element changes as well. We can double each element of an array like this:

```
#!/usr/bin/perl
# forloop2.plx
use warnings;
use strict;

my @array=(10, 20, 30, 40);
print "Before: @array\n";
for (@array) { $_ *= 2 }
print "After: @array\n";
```

This prints:

```
>perl forloop2.plx
Before: 10 20 30 40
After: 20 40 60 80
>
```

If you need to know the number of the element you're currently processing, it's usually best to have the iterator as the range of numbers you're processing – from 0 up to the highest element number in the array. Let's rewrite the joke machine so that it tells *all* the bad jokes, without prompting:

Try It Out – Joke Machine II – The Revenge

Here we use the same jokes tell each of them in turn:

```
#!/usr/bin/perl
# joke2.plx
use warnings;
use strict;

my @questions = qw(Java Python Perl C);
my @punchlines = (
      "None. Change it once, and it's the same everywhere.",
      "One. He just stands below the socket and the world revolves around him.",
      "A million. One to change it, the rest to try and do it in fewer lines.",
      '"CHANGE?!!"'
);

for (0..$#questions) {
      print "How many $questions[$_] ";
      print "programmers does it take to change a lightbulb?\n";
      sleep 2;
      print $punchlines[$_], "\n\n";
      sleep 1;
}
```

The changes to our old `joke1.plx` program produce this result:

```
> perl joke2.plx
How many Java programmers does it take to change a lightbulb?
None. Change it once, and it's the same everywhere.

How many Python programmers does it take to change a lightbulb?
One. He just stands below the socket and the world revolves around him.

How many Perl programmers does it take to change a lightbulb?
A million. One to change it, the rest to try and do it in fewer lines.

How many C programmers does it take to change a lightbulb?
"CHANGE?!!"

>
```

I promise I'll keep my day-job....

How It Works

The for loop is now the main part of our program. Let's have a look at it again:

```
for (0..$#questions) {
     print "How many $questions[$_] ";
     print "programmers does it take to change a lightbulb?\n";
     sleep 2;
     print $punchlines[$_], "\n\n";
     sleep 1;
}
```

The key thing about this example is that we need to match the questions to the punchlines. This means we can't just go through one or the other of the arrays, but we have to go through them both together. We do this by using a list, which counts up from 0 to the highest element of one of the arrays. Since the arrays are both the same size, it doesn't matter which one. The line that does this is:

```
for (0..$#questions) {
```

`$#questions` is the index of the highest element in the `@questions` array. That's different from the value we get when we look at `@questions` in a scalar context. Look:

```
#!/usr/bin/perl
# elems.plx
use warnings;
use strict;

my @array = qw(alpha bravo charlie delta);

print "Scalar value   : ", scalar @array, "\n";
print "Highest element: ", $#array, "\n";
```

```
>perl elems.plx
Scalar value     : 4
Highest element: 3
>
```

Why? There are four elements in the array – so that's the scalar value. Their indices are 0, 1, 2, and 3. Since we're starting at zero, the highest element (`$#array`) will always be one less than the number of elements in the array.

So, we count up from 0 to the index of the highest element in `@questions`, which happens to be 3. We set the iterator to each number in turn. Where's the iterator? Since we didn't give one, perl will use `$_`. Now we do the block four times, once when `$_` is 0, once when it is 1, and so on:

```
        print "How many $questions[$_] ";
```

This line prints the zeroth element of `@questions` the first time around, then the first, then the second, third, and fourth:

```
        print $punchlines[$_], "\n\n";
```

And so it is with the punchlines. If we'd just said:

```
    for (@questions) {
```

`$_` would have been set to each question in turn, but we would not have advanced our way through the answers.

Array Functions

It's time we met some more of the things we can do with arrays. These are variously called **array functions** and **array operators**. As mentioned previously, perl doesn't draw much distinction between functions and operators. The important part is that they all do some kind of work on an array. We've already met one of them: `reverse`, which we used to count down ranges instead of counting up. We can use `reverse` on arrays as well as lists:

```
#!/usr/bin/perl
# countdown.plx
use warnings;
use strict;

my @count = (1..5);
for (reverse(@count)) {
     print "$_...\n";
     sleep 1;
}

print "BLAST OFF!\n";
```

Hopefully, you should have a good idea of what this will print out before you run it.

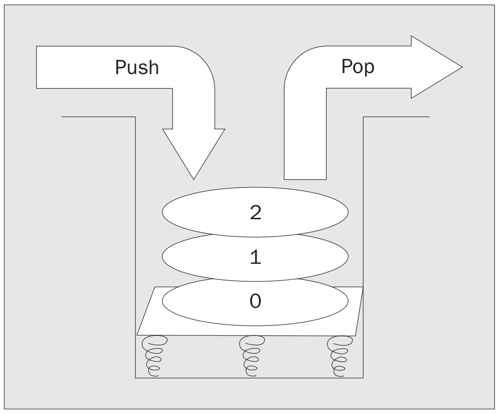
```
>perl countdown.plx
5...
4...
3...
2...
1...
BLAST OFF!
>
```

Now a while back I mentioned some useful functions for adding elements to arrays. Here they are now, along with a couple of other useful tips and tricks.

Pop and Push

Now we've already seen a simple way to add elements to an array: `@array = (@array, $scalar)`.

One of the original metaphors that computer programmers like to use to analyze arrays is a **stack** of spring-loaded plates in a canteen. You push down when you put another plate on the top, and the stack pops up when a plate is taken away:



Following this metaphor, `push` is the operator that adds an element, or list of elements, to the end of an array. Similarly, to remove the top element – the element with the highest index, we use the `pop` operator:

Try It Out – Paper Stacks

Stacks are all around us. In my case, they're all stacks of paper. We can manipulate arrays just as we can manipulate these stacks of paper:

```
#!/usr/bin/perl
# stacks.plx
use warnings;
use strict;

my $hand;
my @pileofpaper = ("letter", "newspaper", "gas bill", "notepad");
```

```
print "Here's what's on the desk: @pileofpaper\n";

print "You pick up something off the top of the pile.\n";
$hand = pop @pileofpaper;
print "You have now a $hand in your hand.\n";

print "You put the $hand away, and pick up something else.\n";
$hand = pop @pileofpaper;
print "You picked up a $hand.\n";

print "Left on the desk is: @pileofpaper\n";

print "You pick up the next thing, and throw it away.\n";
pop @pileofpaper;

print "You put the $hand back on the pile.\n";
push @pileofpaper, $hand;

print "You also put a leaflet and a bank statement on the pile.\n";
push @pileofpaper, "leaflet", "bank statement";

print "Left on the desk is: @pileofpaper\n";
```

Watch what happens:

```
>perl stacks.plx
Here's what's on the desk: letter newspaper gas bill notepad
You pick up something off the top of the pile.
You have now a notepad in your hand.
You put the notepad away, and pick up something else.
You picked up a gas bill.
Left on the desk is: letter newspaper
You pick up the next thing, and throw it away.
You put the gas bill back on the pile.
You also put a leaflet and a bank statement on the pile.
Left of the desk is: letter gas bill leaflet bank statement
>
```

How It Works

Let's take this play-by-play. First off, we initialize our `$hand` and our `@pileofpaper`. Since the pile of paper is a stack, the zeroth element (the letter), is at the bottom, and the notepad is at the top:

```
my $hand;
my @pileofpaper = ("letter", "newspaper", "gas bill", "notepad");
```

We use `pop @array` to remove the top element from the array and it returns that element, which we store in `$hand`. So, we take the notepad from the stack and put it into our hand. What's left? The letter at the bottom of the stack, then the newspaper and gas bill:

```
print "You pick up something off the top of the pile.\n";
$hand = pop @pileofpaper;
print "You have now a $hand in your hand.\n";
```

As we pop again, we take the next element (the gas bill) off the top of the stack and store it again in $hand. Since we didn't save the notepad from last time, it's lost forever now:

```
print "You put the $hand away, and pick up something else.\n";
$hand = pop @pileofpaper;
print "You picked up a $hand.\n";
```

The next item is the newspaper. We pop this as before, but we never store it anywhere:

```
print "You pick up the next thing, and throw it away.\n";
pop @pileofpaper;
```

We've still got the gas bill in $hand from previously. push @array, $scalar will add the scalar onto the top of the stack. In our case, we're putting the gas bill on top of the letter:

```
print "You put the $hand back on the pile.\n";
push @pileofpaper, $hand;
```

push can also be used to add a list of scalars onto the stack – in this case, we've added two more strings. We could add the contents of an array to the top of the stack with push @array1, @array2. So we now know that we can replace a list with an array:

```
print "You also put a leaflet and a bank statement on the pile.\n";
push @pileofpaper, "leaflet", "bank statement";
```

As you might suspect, you can also push lists of lists onto an array: They simply get flattened first into a single list and then added.

Shift and Unshift

While the functions push and pop deal with the 'top end' of the stack, adding and taking away elements from the highest index of the array – the functions unshift and shift do the corresponding jobs for a similar job for the bottom end:

```
#!/usr/bin/perl
#shift.plx
use warnings;
use strict;

my @array = ();
unshift(@array, "first");
print "Array is now: @array\n";
unshift @array, "second", "third";
print "Array is now: @array\n";
shift @array ;
print "Array is now: @array\n";
```

```
>perl shift.plx
Array is now: first
Array is now: second third first
Array is now: third first
>
```

First we `unshift()` an element onto the array, and the element appears at the beginning of the list. It's not easy to see this since there are no other elements, but it does. We then `unshift` two more elements. Notice that the entire list is added to the beginning of the array all at once, rather than one element at a time. We then use `shift` to take off the first element, ignoring what it was.

Sort

One last thing you may want to do while processing data is put it in alphabetical or numeric order. The `sort` operator takes a list and returns a sorted version:

```
#!/usr/bin/perl
#sort1.plx
use warnings;
use strict;

my @unsorted = qw(Cohen Clapton Costello Cream Cocteau);
print "Unsorted: @unsorted\n";
my @sorted = sort @unsorted;
print "Sorted:   @sorted\n";
```

```
>perl sort1.plx
Unsorted: Cohen Clapton Costello Cream Cocteau
Sorted:   Clapton Cocteau Cohen Costello Cream
>
```

This is only good for strings and alphabetic sorting. If you're sorting numbers, there is a problem. Can you guess what it is? This may help:

```
#!/usr/bin/perl
#sort2.plx
use warnings;
use strict;

my @unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
my @sorted = sort @unsorted;
print "Sorted:   @sorted\n";
```

```
>perl sort2.plx
Sorted:   1 11 2 24 3 36 4 40
>
```

What? 11 doesn't come between 1 and 2. What we need to do is compare the numeric values instead of the string ones. Cast your mind back to last chapter and recall how to compare two numeric variables, `$a` and `$b`. Here, we're going to use the `<=>` operator. `sort` allows us to give it a block to describe how two values should be ordered, and we do this by comparing `$a` and `$b`.These two variables are given to us by the `sort` function:

```
#!/usr/bin/perl
#sort3.plx
use warnings;
use strict;
my @unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
```

```
   my @string = sort { $a cmp $b } @unsorted;
   print "String sort:  @string\n";

   my @number = sort { $a <=> $b } @unsorted;
   print "Numeric sort:  @number\n";
```

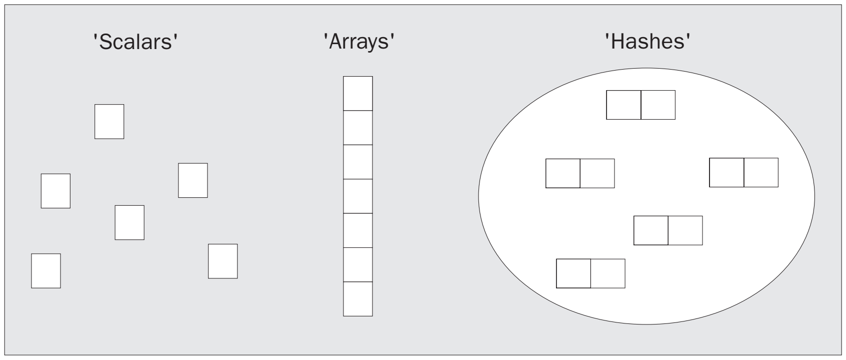
```
>perl sort3.plx
String sort:  1 11 2 24 3 36 4 40
Numeric sort:  1 2 3 4 11 24 36 40
>
```

Another good reason for using string comparison operators for strings and numeric comparison operators for numbers!

Hashes

The final variable type we have is the hash. In the introduction, I said that the hash was like a dictionary or a phone book, but that's not quite right. There is a slight difference in that a phone book is normally ordered – the names are sorted alphabetically. In a hash the data is totally unsorted and has no intrinsic order. In fact, it's more like directory enquiries than a phone book in that you can easily find out what the number is if you have the name. Someone else keeps the order for you, and you needn't ask what the first entry is.

Here's where a diagram helps:



A scalar is one piece of data; it's like a single block. An array or a list is like a tower of blocks: it's kept in order, and it's kept together as a single unit. A hash, on the other hand, is more like the diagram on above. It contains several pairs of data. The pairs are in no particular order (no pair is 'first' or 'top'), and they're all scattered around the hash.

Creating a Hash

A hash looks very similar to a list, and it also behaves very much like a list. It's only actually effective as a hash when you store it in a hash variable. Just like scalar variables have a `$` prefix, and arrays have a `@` prefix, hashes have their own prefix – a percent sign `%`. Again, the same naming rules apply, and the variables `%hash`, `$hash`, and `@hash` are all different.

There are two ways of writing a hash. First, just like an ordinary list of pairs:

```
%where=(
        "Gary"     , "Dallas",
        "Lucy"     , "Exeter",
        "Ian"      , "Reading",
        "Samantha" , "Oregon"
);
```

In this case, the hash could be saying that "Gary's whereabouts is Dallas", "Lucy lives in Exeter" and so on. All it really does is pair Gary and Dallas, Lucy and Exeter, and so on. How the pairing is interpreted is up to you.

If we want to make the relationship a little clearer, as well as highlighting the fact that we're dealing with a hash, we can use the => operator. That's not >=, which is greater-than-or-equal-to; the => operator acts like a 'quoting comma'. Essentially, it's a comma, but whatever appears on the left hand side of it – and only the left – is treated as a double-quoted string:

```
%where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

The scalars on the left of the arrow are called the **hash keys**, the scalars on the right are the **values**. We use the keys to look up the values:

Hash keys must be unique. You cannot have more than one entry for the same name, and if you try to add a new entry with the same key as an existing entry, the old one will be over-written. Hash values meanwhile need not be unique.

Key uniqueness is more of an advantage than a limitation. Every time the word 'unique' comes into a problem, like counting the unique elements of an array, your mind should immediately echo 'use a hash!'

Because hashes and arrays are both built from structures that look like lists, you can convert between them, from array to hash like this:

```
@array = qw(Gary Dallas Lucy Exeter Ian Reading Samantha Oregon);
%where = @array;
```

And then back to an array, like so:

```
@array = %where;
```

However, you need to be careful when converting back from a hash to an array. Hashes do not have a guaranteed order. Although values will always follow keys, you cannot tell what order the keys will come in. Since hash keys are unique, however, we can be sure that `%hash1 = %hash2` will copy a hash accurately.

If you need to turn your hash around, to look up people by location, you can use this list-like structure to your advantage... just `reverse` the list. Be careful though – if you have two values that are the same, then converting them to keys means that one will be lost. Remember that keys must be unique:

```
@array = qw(Gary Dallas Lucy Exeter Ian Reading Samantha Oregon);
%where = @array;
```

`%where` now holds the same value as if the following call had been made:

```
%where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

Likewise, `%who` will hold the same values no matter which of the two calls below were made:

```
%who = reverse @array;
%who = (
     Oregon  => "Samantha",
     Reading => "Ian",
     Exeter  => "Lucy",
     Dallas  => "Gary"
);
```

Working with Hash Values

To look up a value in a hash, we use something similar to the index notation for arrays. However, instead of locating elements by number, we're now locating them by name; instead of using square brackets, we use braces (curly brackets):

Try It Out – Using Hashes

Here's a simple example of looking up details in a hash:

```
#!/usr/bin/perl
#hash1.plx
use warnings;
use strict;

my $place = "Oregon";
my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
my %who = reverse %where;

print "Gary lives in ", $where{Gary}, "\n";
print "Ian lives in $where{Ian}\n";
print "$who{Exeter} lives in Exeter\n";
print "$who{$place} lives in $place\n";
```

```
> perl hash1.plx
Gary lives in Dallas
Ian lives in Reading
Lucy lives in Exeter
Samantha lives in Oregon
>
```

How It Works

First, we set up our main hash, which tells us where people live:

```
my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);
```

By reversing the order of the list, we produce a hash that tells us who lives where:

```
my %who = reverse %where;
```

When doing this you need to be careful, as I have already mentioned. You must not have two values the same, since they will need to become keys, and keys must be unique – one or other of them will get lost.

Now we can look up an entry in our hashes – we'll ask 'Where does Gary live?':

```
print "Gary lives in ", $where{Gary}, "\n";
```

This is almost identical to looking up an array element, except for the brackets and the fact that we are now allowed to use strings as well as numbers to index our elements.

```
print "Ian lives in $where{Ian}\n";
print "$who{Exeter} lives in Exeter\n";
```

The braces of a hash look-up can also quote what is inside them in double quotes if we do not provide the quotes ourselves:

```
print "$who{$place} lives in $place\n";
```

Just as with array elements, we need not use a literal to index the element – we can look-up using a variable as well.

Adding, Changing, and Taking Values Away from a Hash

Hash entries are very much like ordinary scalar variables, except that you need not declare an individual hash key before assigning to it or using it. We can add a new person to our hash just by assigning to their hash entry:

```
$where{Eva} = "Uxbridge";
print "Eva lives in $where{Eva}\n";
```

A new entry springs into existence, without any problems. We can also change the entries in a hash just by reassigning to them. Let's move people around a little:

```
$where{Eva}      = "Denver";
$where{Samantha} = "California";
$where{Lucy}     = "Tokyo";
$where{Gary}     = "Las Vegas";
$where{Ian}      = "Southampton";

print "Gary lives in $where{Gary}\n";
```

To remove an entry from a hash, you need to use the `delete()` function, as we do in this little variant on `hash1.plx`:

```
#!/usr/bin/perl
#badhash1.plx
use warnings;
use strict;

my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);

delete $where{Lucy};
print "Lucy lives in $where{Lucy}\n";
```

Now here we delete Lucy's entry in `%where` before we access it. So after running it, we should get an error. Sure enough, we get:

```
> perl badhash1.plx
Use of uninitialized value in concatenation (.) at badhash1.plx line 11
Lucy lives in Exeter
>
```

It's not that we haven't initialized poor Lucy, but rather that we've decided to get rid of her.

Accessing Multiple Values

The problem with hashes looking like lists is that we can't really use for loops on them directly. If we did, we would get both keys and values with no indication as to which was which. To help us, Perl provides three functions for iterating over hashes.

First, there is `keys (%hash)`. This gives us a list of the keys (all of the scalars on the left-hand side). This is usually what we want when we wish to visit each hash entry in turn:

Try It Out – Looping Over A Hash

Let's leave the computer to run over hash and tell us where each person lives:

```
#!/usr/bin/perl
#hash2.plx
use warnings;
use strict;

my %where=(
        Gary     => "Dallas",
        Lucy     => "Exeter",
        Ian      => "Reading",
        Samantha => "Oregon"
);

for (keys %where) {
     print "$_ lives in $where{$_}\n";
}
```

Currently, this tells me:

```
>perl hash2.plx
Samantha lives in Oregon
Gary lives in Dallas
Lucy lives in Exeter
Ian lives in Reading
>
```

You may find that the output appears in a different order on your machine. Don't worry, as I said, hashes are unordered, and there's no guarantee that the keys will come out in the same order each time. It really depends on the particular version of Perl that you are using.

How It Works

Here is the part that does all the work:

```
for (keys %where) {
     print "$_ lives in $where{$_}\n";
}
```

`keys` is a function which, like `sort` and `reverse`, returns a list. The list in my case was `qw(Samantha Gary Lucy Ian)`, and `for` visited each of those values in turn. As `$_` was set to each one, we could print the name and look up that entry in the hash.

The counterpart to `keys` is `values`, which returns a list of all of the values in the hash. This is somewhat less useful, since you can always find the value if you have the key, but you cannot easily find the key if you have the value. It's almost always advantageous to use `keys` instead.

The final function is `each`, which we will look at later. It returns each hash entry as a key-value pair.

Summary

Lists are a series of scalars in order. Arrays are variable incarnations of lists. Both lists and arrays are flattened, so we cannot yet have a distinct list inside another list. We get at both lists and arrays with square-bracket subscripts. These can be single numbers, or a list of elements. If we're looking up a single scalar in an array, we need to remember to use the form `$array[$element]`, because the variable prefix always refers to what we want, not what we have got. We can also use ranges to save time and to specify list and array slices.

Perl differentiates between scalar and list context and returns different values depending on what the statement is expecting to see. For instance, the scalar context value of an array is the number of elements in it; the list context value is of course the list of the elements themselves.

Hashes are unordered structures made up of pairs, each pair consisting of a key and a value. Given the key, we can look up the entry. Generally, `$hash{$key} = $value`. We can loop over all the elements of a list or array using a for loop. We need to modify this when looping over two lists at once or when looking for the keys or values of a hash.

Exercises

1. When you assign to a list, the elements are copied over from the right to the left:

   ```
   ($a, $b) = ( 10, 20 );
   ```

   will make `$a` become 10 and `$b` become 20. Investigate what happens when:

   - ❑ There are more elements on the right than on the left.
   - ❑ There are more elements on the left than on the right.
   - ❑ There is a list on the left but a single scalar on the right.
   - ❑ There is a single scalar on the left but a list on the right.

2. What elements make up the range `('aa' .. 'bb')`? What about `('a0' .. 'b9')`?

3. Store your important phone numbers in a hash. Write a program to look up numbers by the person's name.

4. Turn the joke machine program from two arrays into one hash. While doing so, write some better lightbulb jokes.